THE IN-SYSTEM CONFIGURATION HANDBOOK: *A DESIGNER'S GUIDE TO ISC*

Neil G. Jacobson

THE IN-SYSTEM CONFIGURATION HANDBOOK:

A Designer's Guide to ISC

THE IN-SYSTEM CONFIGURATION HANDBOOK:

A Designer's Guide to ISC

by

Neil G. Jacobson
Xilinx, U.S.A.

KLUWER ACADEMIC PUBLISHERS
Boston / Dordrecht / New York / London

Distributors for North, Central and South America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA
Telephone (781) 871-6600
Fax (781) 871-6528
E-Mail <kluwer@wkap.com>

Distributors for all other countries:
Kluwer Academic Publishers Group
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS
Telephone 31 78 6576 000
Fax 31 78 6576 474
E-Mail <orderdept@wkap.nl>


Electronic Services <http://www.wkap.nl>

Library of Congress Cataloging-in-Publication

CIP info or:

Title: The In-System Configuration Handbook: A Designer's Guide to ISC
Author (s): Neil G. Jacobson
ISBN: 1-4020-7655-X

Printed on acid-free paper.

Printed in the United States of America

Dedication

To Dina and Joseph whom I love

Table of Contents

Preface

Programmable logic radically changed the electronic system design landscape. It reduced board space needed for random logic, state machines and system interfaces. It allowed faster design cycles, made easy late term bug fixes and gave designers greater freedom to experiment and prototype.

In-system programming of these devices has had a similar revolutionary effect. The ability to change the programmed content of programmable logic while it is on the board is equivalent to being able to redesign all the hardware - without changing a single component.

This allows the possibility of providing field upgrades of your product to fix problems or to introduce new functionality. It allows designing in reconfiguration as an essential function of your system with different capabilities swapped in as needed during run-time. Further it allows storage of different product profiles for retrieval as necessary to allow just-in-time configuration of systems to meet market needs.

Recent developments in programmable logic have helped in making realizing reconfigurable systems more streamlined. The most significant development, though, was the introduction, approval and popularization of IEEE STD 1532, the IEEE Standard for In-System Configuration of Programmable Devices.

The purpose of this text is to bring together, in a single volume, the information needed by systems designers to develop applications that include configurability. This covers the entire range of systems from the

simplest implementations that merely include configurable logic to realize system functions to the most complicated that include reconfigurability as part of the application itself.

While focusing on IEEE STD 1532, the text surveys all the available techniques and products that ease developing in-system configurable applications. In addition, we detail design considerations and rules-of-thumb to ensure the functionality you want will work.

The book begins with a historical perspective on programmable logic. Understanding where you have been often clarifies the present and sheds light in the future. Then we will examine the architecture of programmable logic devices, surveying the most popular devices. From that basis, we will look into the programmable technology at the core of the devices and understand how that works.

After understanding the hardware we are working with, we will survey the infrastructure support provided with these devices. By this, we are referring to the files used to provide programming data for the device. It is here that we gain knowledge of IEEE STD 1532.

From there we study the characteristics of IEEE STD 1532 devices and then begin the analysis of in-system configurable application design. We look into the types of tools available to help you in completing your system and the applicable system design rules. We end with an exploration of the many types of configurable systems and guidelines for their construction.

The object is for this book to be both useful and practical in nature and serve as a reference for developing in-system configurable systems of the present and the future.

Acknowledgments

I would like to thank my reviewers C. J. Clark, Dave Bonnett, Vince Eck, Dennis Lia, Mark Moyer, Ken Parker, and Jesse Jenkins. Your exceptional efforts and helpful feedback contributed substantially to this book.

Special thanks to the patient Carl Harris at Kluwer Academic Publishers, who I am certain, never thought he would see this text completed.

I would also like to thank my loving wife, Dina, and my son, Joseph for allowing me to pursue this craziness during what would have otherwise been "our time".

Chapter 1

A Brief History of In-System Configuration

1. Background

Programmable logic grew from the humble beginnings of Programmable Logic Arrays (PLA) and through Programmed Array Logic (PAL), to Programmable Logic Devices (PLD) and Field Programmable Gate Arrays (FPGA).

Each step in the development increased the speed, flexibility, complexity and capabilities of the devices. As well, the prices decreased. This typical technological evolution led to increasing acceptance and use of the programmable logic.

Worthy of emphasis, though, is that these devices are programmable. They do nothing until programmed with the design personality the end user needs. Early on, the primary purpose of programmability was to get the device working. This wasn't surprising. Programming was complicated and unreliable and typically carried out only once. Soon, the nature of the programmable cell at the heart of these devices allowed for simpler programming techniques. As well, easy reprogramming was possible. Programming then simplified to the point in which the device itself was responsible its own programming. There was no need for external special purpose hardware. This, in turn, led to developing in-system configuration. With in-system configuration, end users could begin to examine the utility of reconfiguration of device contents as an essential part of the system. This is the premise of this book.

2. Proprietary Approaches

Some PLDs first incorporated in-system configuration because of the process technology adopted. Manufacturers did not see this as a key selling

point but a means to an end. Later, others developed it as a product differentiator and used it as a key selling point.

Static read-only memory (SRAM) cell based devices were always in-system configurable. Since the devices had a volatile data store, there needed to be a method to get the configuration bits into the device. Since the technology was new and no applicable standards existed, manufacturers opted for proprietary configuration ports. Typically, manufacturers provided two methods. The first, a serial port, accepted data from a serial programmable read-only memory (SPROM). The second was a parallel port, typically used with a microprocessor or special control logic to load configuration data 8 bits at a time.

Both approaches introduced their own protocols. SPROMs created a new market segment to supply turnkey devices that incorporated the control protocol with a PROM on a single chip. Publishing the protocol allowed end users to fashion their own SPROMs, using some control logic (typically a CPLD) and an off-the-shelf parallel PROM.

An inexpensive, simple microprocessor could use the parallel protocol to do fast and intelligent loading of multiple SRAM devices. This would allow users to manage and optimize the configuration method and configuration store.

The serial protocol was simple and needed fewer device pins. This supported designs with a larger configuration time budget and a greater need for more input and output pins (IO), as well. Using a microprocessor as a configuration controller driving the serial port is also possible.

It wasn't long before users connected the configuration port access and the characteristic reprogrammability of the devices together to incorporate reconfigurability into their designs.

For nonvolatile devices, the path was longer. Early nonvolatile technologies needed special voltages to the device to program the contents. The programming voltages were higher than the typical system voltages of 5 volts. Sometimes, the algorithm needed voltage pulsing with significant pulse-width accuracy to program the device correctly. These special requirements forced the use of special purpose machines known as device programmers to get a device configured. This created a new application for an industry that was already serving the ROM and PROM market. The devices to be configured were inserted in a socket on the device

programmer. An operator would select the programming source file and direct the machine to configure the device with the file's contents. The development of special purpose device handling hardware and special gang programmers increased throughput and fostered integration of this approach into manufacturing flows. Device handlers could pick up a device, insert it in a programmer, retrieve it after configuration, and then place it on the target board for soldering. Gang programmers could program a large group of similar devices with the same data concurrently to increase the programming rate.

Every device had a different algorithm and different voltage needs. Companies that developed device programmers struggled to keep up-to-date with their end user needs.

2.1.1 Lattice Semiconductor and In-System Programming

Process technology advanced and the device geometries shrank. The shrinking feature size allowed for two developments. First, the voltage needed to program nonvolatile cells was reduced. Second, the available die area increased for integration of programming control logic and the generation of on-chip programming voltages. This made the developing in-system configuration possible.

Lattice Semiconductor introduced what they called "In-System Programming" in 1996. A simple four pin serial interface for configuration conserved the number of IO pins needed. The four pins are:

- SDI (serial data input)
- MODE
- SCLK (serial clock)
- SDO (serial data output)

These four pins supply programming data to the device and drive an underlying controlling state machine that configures the device.

The SDI pin performs two different roles. First, it acts as the data input to the serial shift register built inside the device. Second, it serves as one of two control pins for the programming state machine. Because of this dual role, the MODE pin controls the role of SDI. When MODE is low, SDI becomes the serial input to the shift register. When MODE is high, SDI becomes a control signal for the programming state machine.

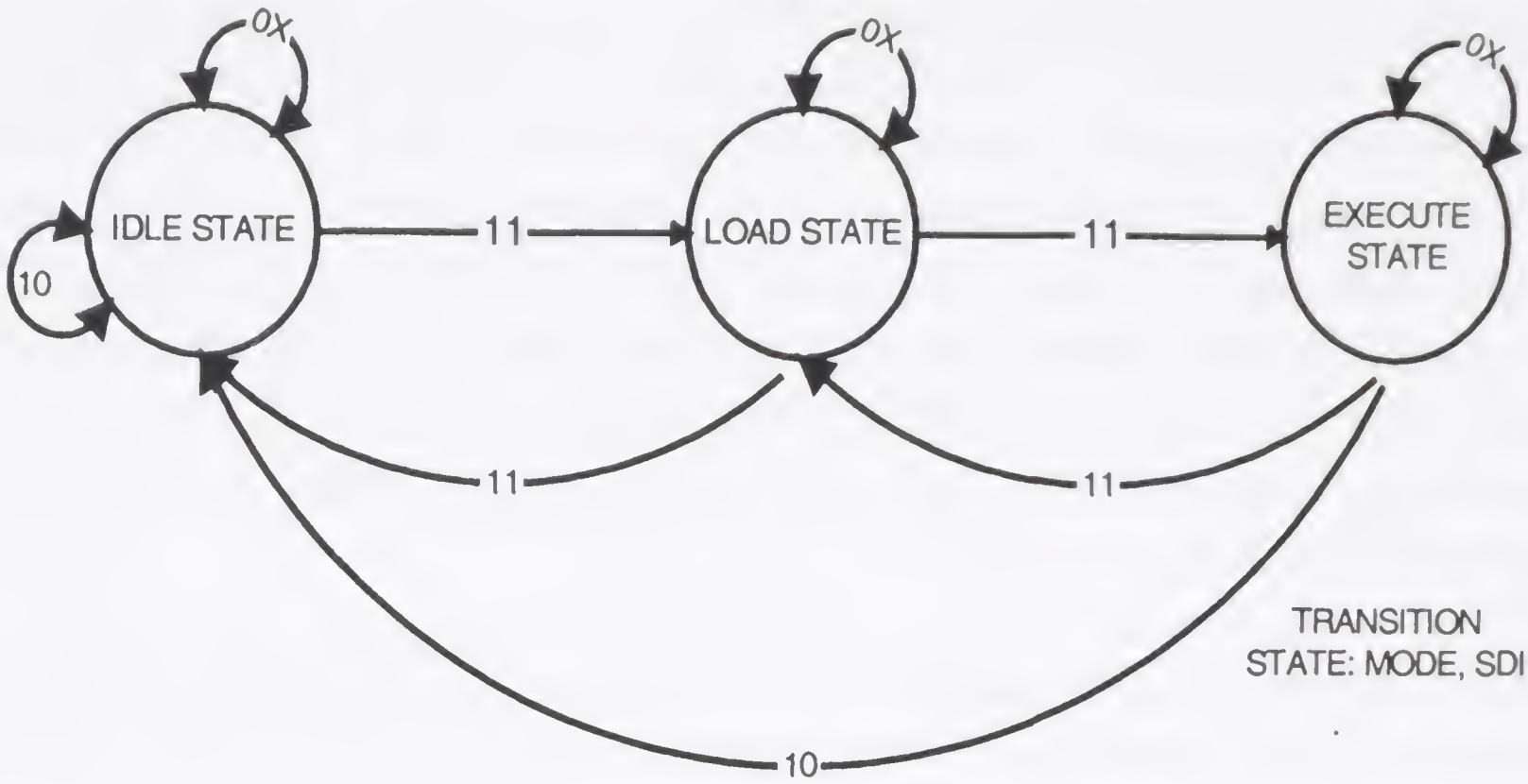

Figure 1-1. Lattice Semiconductor Programming State Machine

This means the MODE signal, combined with the SDI signal, controls the programming state machine.

The SCLK pin provides the serial shift register with a clock. SCLK clocks the internal serial shift registers and clock the programming state machine between states.

The SDO pin connects to the output of the internal serial shift registers. When MODE is high, SDO connects directly to SDI, bypassing the device's shift registers.

The state machine consists of three states: **Idle**, **Load** and **Execute**. The values of SDI and MODE at the rising edge of SCLK control the state transitions. When powered, the device wakes in the **idle** state. To run a configuration program, the device transitions to the **load** state to load the instructions and data and then to the **execute** state to complete the operation.

The protocol allowed daisy chaining of Lattice devices and was optimized for use with Lattice Semiconductor's programming algorithms. While this approach provided great utility to users of these specific devices, the protocol was proprietary and Lattice Semiconductor was not keen to share it.

It turned out that was all right since another standard was starting to come into use that was an obvious choice for in-system configuration.

3. Standard Approaches

In 1985, a group of European test engineers and technologists gathered to discuss challenges and costs associated with board test. The complexity and price of the automatic test equipment (ATE) because of shrinking packages and falling voltages challenged manufacturers of sophisticated electronics. This group began to discuss ways in which to amend the silicon to include certain testability circuits to offload complexity from the ATE to the device. This group became the Joint European Test Action Group or JETAG.

In 1988, the JETAG engaged engineers from North America in their discussions. This lead to dropping the "E" and thus the Joint Test Action Group or JTAG arrived. This group developed the early proposal for a boundary-scan standard. The standardization was carried out with the backing of the Institute of Electrical and Electronics Engineers (IEEE). In 1990, the IEEE formally approved and published the first boundary-scan standard, known as IEEE STD 1149.1.

3.1 IEEE STD 1149.1

Boundary-scan technology enables engineers to perform extensive debugging and diagnostics on a system through four dedicated test pins. Signals are scanned into and out of the IO cells of a device serially to control its inputs and test the outputs under various conditions.

Devices that support IEEE STD 1149.1 contain a shift-register cell for each signal pin of the device. These register cells are connected in a dedicated path around the device's boundary. Together these cells are known as the boundary-scan register. This register creates an access path that avoids the normal inputs and provides direct control of the device and detailed visibility at its outputs. Access to and manipulations of this register are controlled by the four test pins and their associated control logic.

The four boundary-scan control signals, collectively referred to as the Test Access Port (TAP), define a serial protocol port for boundary-scan based devices. The pins are as follows:

- TCK - (Test Clock) - synchronizes the internal state machine operations.

- TMS - (Test Mode Select) - sampled at the rising edge of TCK to determine the next state.
- TDI - (Test Data Input) - sampled at the rising edge of TCK and shifted into the device's test logic when the internal state machine is in the correct state.
- TDO - (Test Data Output) - represents the data shifted out of the device's test logic and is valid on the falling edge of TCK when the internal state machine is in the correct state.

The standard also allows an optional fifth pin called TRST (Test logic Reset). When driven low, this signal asynchronously resets the internal state machine. Because there exists a synchronous method to reset the state machine using the other pins, most IEEE STD 1149.1 devices do not include TRST.

The TCK and TMS (and TRST) input pins drive a 16-state TAP controller state machine. The TAP controller manages the exchange of data and instructions. The controller advances to the next state based on the value of the TMS signal at each rising edge of TCK.

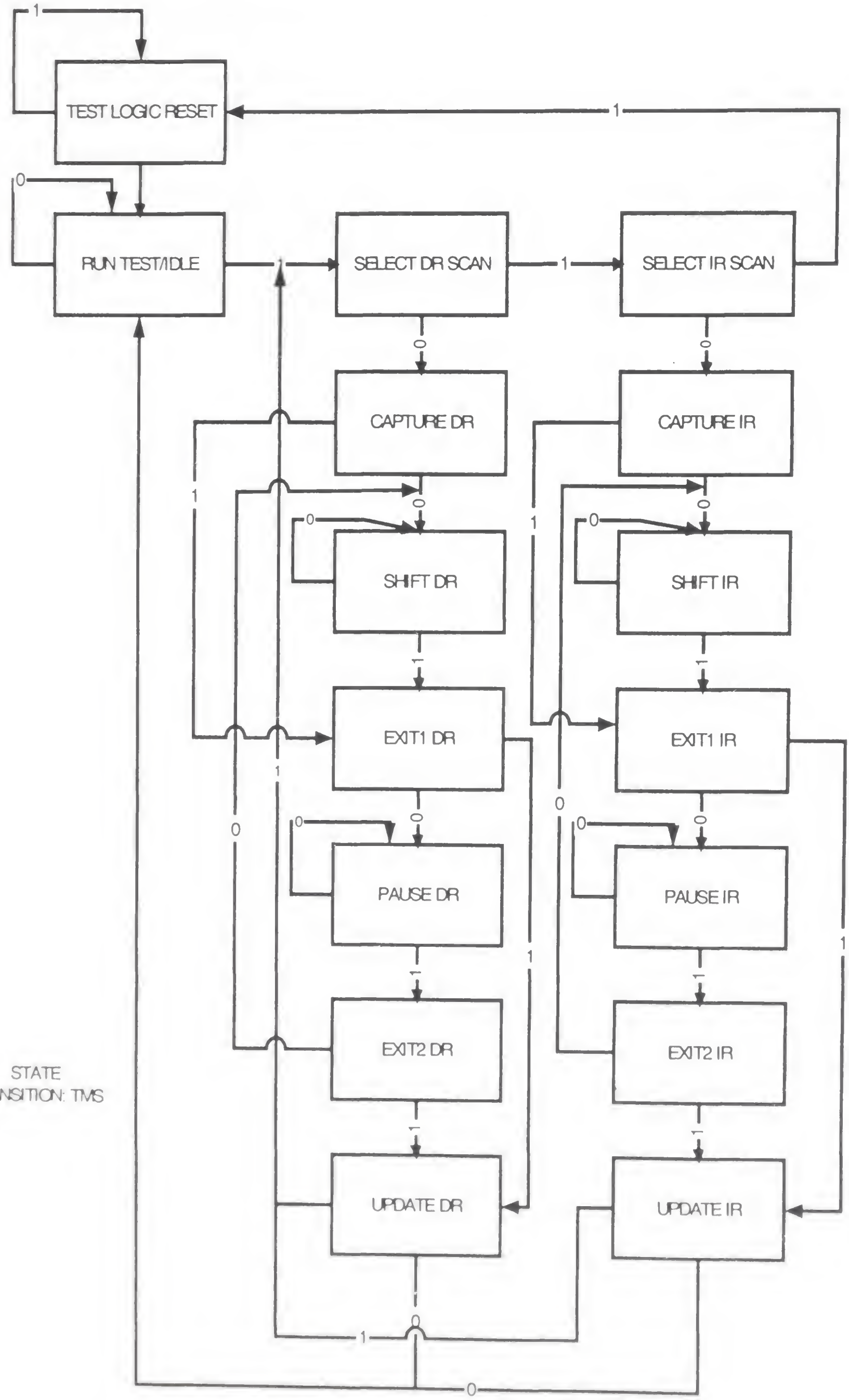

Figure 1-2. TAP Controller State Machine

The sixteen states of the TAP controller state machine are as follows:

- **Test Logic Reset** – You arrive at this state by holding TMS high for five TCK pulses. This resets the logic of the TAP controller
- **Run Test/Idle** – Operations execute in this state after the associated data has been loaded or simply to wait for signals to settle before sampling them or capturing them.
- **Select DR Scan** – This transitional state leads to either data register operations or instruction register operations.
- **Capture DR** – This state loads the selected data register with values typically sampled from the device's pins or from some internal device states. The active instruction defines the behavior.
- **Shift DR** – TDI sampling occurs in this state. In the state, the TAP controller connects a data register between TDI and TDO of length and type determined by the active instruction. With each rising edge of TCK, data shifts into the register from TDI and shifted out on TDO.
- **Exit1 DR** – This transitional state leads to either the Pause DR or Update DR state.
- **Pause DR** – This state allows the hardware controlling the TAP a method to break shifts up into smaller bit chunks to ease the processing burden. After completing the pause, the Shift state may be reentered.
- **Exit2 DR** – This is a transitional state that leads either to the Shift DR or Update DR state.
- **Update DR** - The state takes the data loaded in the shift register in the Shift DR state and loads it into he active electronics of the device.
- **Select IR Scan** – This transitional state leads to either instruction register operations or the Test Logic Reset state.
- **Capture IR** - This state loads the instruction register with values defined by the standard.
- **Shift IR** - TDI sampling occurs during this state. In the state, the TAP controller connects the fixed length instruction register between TDI and TDO. With each rising edge of TCK data shifts into the register from TDI and shifted out on TDO.
- **Exit1 IR** - This transitional state leads to either the Pause IR or Update IR state
- **Pause IR** - This state allows the hardware controlling the TAP, a method to break shifts up into smaller bit chunks to ease the processing burden. After completing the pause, the Shift state may be reentered.

- **Exit2 IR** - This is a transitional state that leads either to the Shift IR or Update IR state
- **Update IR** - The state takes the instruction loaded in the shift register in the Shift IR state and loads it to make it become the active instruction.

Each device has only one instruction register. The instruction register length is fixed. Every instruction must have a data register associated with it. The two main paths in the state transition diagram are the DR path and the IR path. The DR path controls the operations on the data registers. The IR path control operations on the instruction register. The data register selected through the DR path is based on the instruction loaded in the instruction register after traversing the IR path.

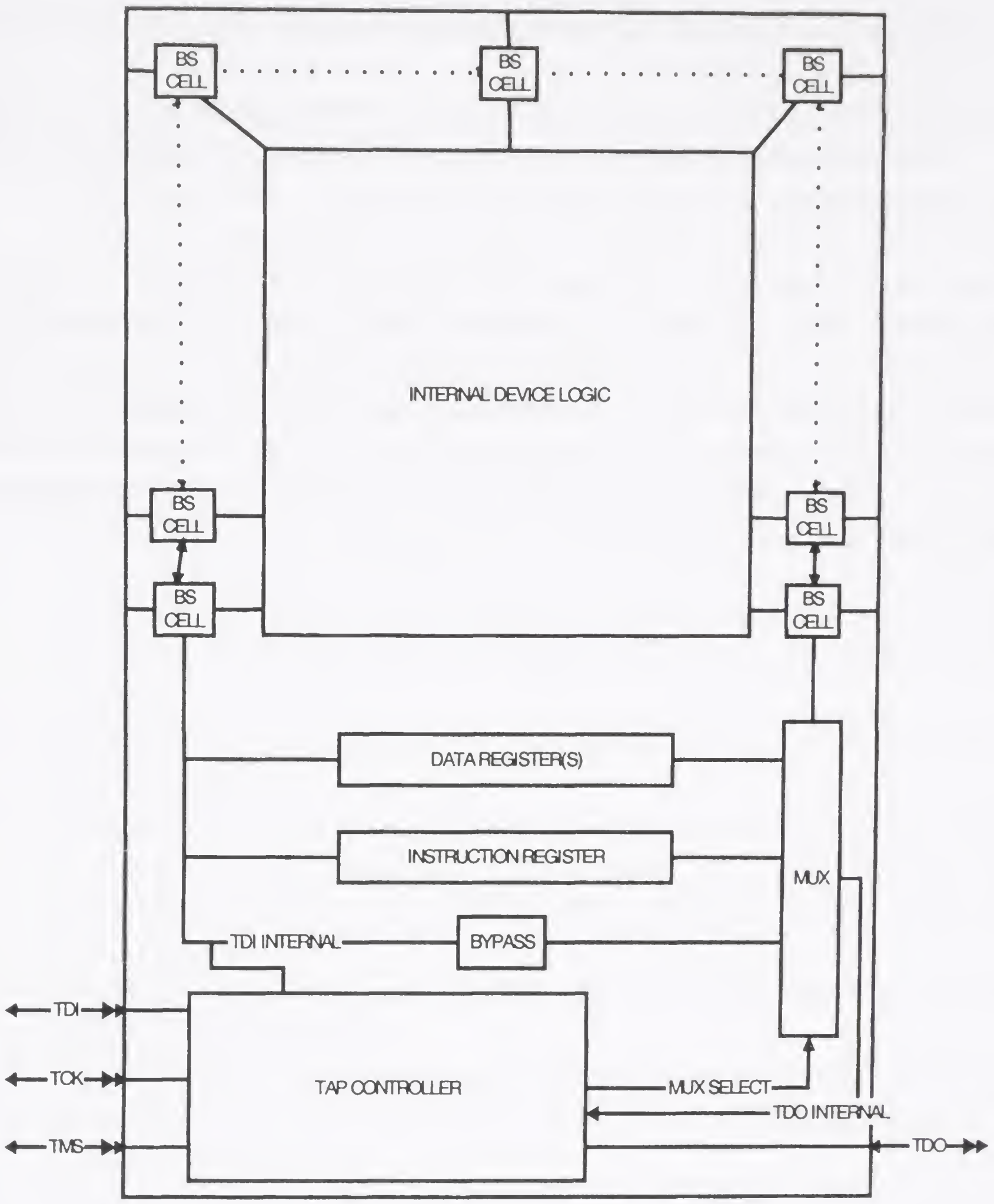

Figure 1-3. Block Diagram of an IEEE STD 1149.1 Compliant Device

A transition path like the following loads a new value into the Instruction Register:

1. Run Test/Idle
2. Select DR Scan
3. Select IR Scan
4. Capture IR
5. Shift IR (shift in instruction bits one at a time)
6. Exit1 IR (last instruction bit shifted in)
7. Update IR (instruction shifted in now the active instruction)

8. Run Test/Idle

Now with an instruction loaded and active, you can load the data needed by the instruction, into its associated data register. A transition path like the following loads a new value into this data register:

1. Run Test/Idle
2. Select DR Scan
3. Capture DR
4. Shift DR (shift in data bits one at a time)
5. Exit1 DR (last data bit shifted in)
6. Update DR (data shifted in now loaded into the device electronics)
7. Run Test/Idle

As the new value shifts into the currently selected Data Register on TDI, the captured value shifts out on TDO.

The following data-registers are present in every IEEE STD 1149.1 compliant device:

- The *Bypass* register - A 1 bit pass-through register that connects the TDI to the TDO with a 1-clock delay to give access to another device in the daisy chain on the same board.
- The *Boundary Scan* register (BSR) – this register intercepts all the signals between the core-logic and the pins and drives the interconnect tests.

IEEE STD 1149.1 defines a compulsory set of instructions that must be present in all compliant implementations. This compulsory set contains the following instructions:

- *BYPASS*: When active, this instruction connects the single bit BYPASS register between TDI and TDO.
- *EXTEST*: When active, this instruction connects the boundary scan register between the TDI and TDO. The device's pin states are sampled and captured by the BSR cells in the Capture DR state. The captured contents of the BSR shift out TDO as new values shift in on TDI in the Shift DR state. The new BSR values are applied to the chip's pins in the Update DR state

The normal sequence used to perform a test operation is:

1. Load an instruction that specifies the test performed (say, EXTEST).
2. Load the Data Register with values used during this test.
3. Optionally, go to Run Test/Idle to wait for applied values to settle.
4. Load the Data Register with the next values, while collecting the results of the previous values applied.
5. Repeat from step 3 until all values are exhausted.

That represents a quick summary of the features of the boundary-scan standard suitable for interconnect test. This was its primary and intended application.

In 1993, improvements and corrections to the standard were approved. Following that, 1994 saw the approval of a standard language for describing the boundary-scan capacities of an IEEE STD 1149.1 compliant device. This language, known as Boundary-Scan Description Language (BSDL), is input to boundary-scan tools to allow them to understand the manner in which to use a compliant device automatically.

By 1994, the complete boundary-scan infrastructure was available. A well-defined hardware standard was approved and a well-defined boundary-scan capability description language was available. This language was Boundary-Scan Description Language (BSDL). We will learn more about this later. Concurrent with this, device geometries were shrinking and speed and area overhead associated with the test electronics of IEEE STD 1149.1 were acceptable.

This set the stage for the broader adoption of IEEE STD 1149.1 as a device test standard. The true power of the standard, however, was that it defined an extensible architecture. Once the TAP was in place with its associated state machine, there were no limits on defining instructions, data registers, or functions supported.

As the adoption of IEEE STD 1149.1 increased, it made little sense to have a separate proprietary port dedicated to in-system configuration and one for boundary-scan test. Integration of the functionality became certain. This was possible owing to the extensibility of the IEEE STD 1149.1 architecture. Vendors rushed to set up in-system configuration within IEEE STD 1149.1. Each vendor worked alone and developed similar but rather different approaches. Therefore, while the devices could be connected to

one another on an IEEE STD 1149.1 daisy chain, there were incompatibilities.

Some devices did not fulfill all of IEEE STD 1149.1, choosing to neglect the boundary-scan test functionality. This left their devices as an interconnect test hole on the board. Other manufacturers used the IEEE STD 1149.1 state machine transitions in an unusual way during programming. This needed special processing for those devices that could harm other devices that had different schemes. Still other devices had unusual or unspecified IO behavior before and during configuration that forced special handling.

The rush to IEEE STD 1149.1 was a hopeful first step. However, it did not reduce the need for customized vendor-specific solutions.

Chapter 2

CONFIGURABLE DEVICE ARCHITECTURES

1. Introduction

Programmable logic is an ideal medium for customized digital designs. Like microprocessors and memories, it offers the well-known advantages of high integration: high complexity and density, small size, low power consumption and cost, and high reliability. Programmable logic also avoids all the problems associated with Application Specific Integrated Circuits (ASIC):

- High Non-Recurring Engineering (NRE) costs (such as those charges associated with mask fabrication)
- Inventory management costs
- Long delays in development and fabrication
- Complex testing issues
- Design issues related to deep sub-micron design rules

This might make programmable logic seem like the only reasonable solution for almost any application. However, some disadvantages have yet to be overcome. For instance, the high cost of high-density programmable devices when compared to similar sized ASICs and the inability of programmable logic to meet the speeds of ASICs. The programmable logic community is rapidly addressing these disadvantages. Before we examine the issues related to the mechanics of configuring programmable devices, let us first get a better understanding of the variety of programmable devices on the market.

2. Programmable Logic Architectures

As with most technologies, programmable logic has changed significantly since its first introduction thirty years ago. Understanding this

evolution helps shed light on today's situation. In this section, we will provide a survey the architectural evolution of programmable logic devices (PLD) from Simple Programmable Logic Devices (SPLD) to Complex Programmable Logic Devices (CPLD) to Field Programmable Gate Arrays (FPGA).

Table 2-1. Gate Capacity for Device Categories

Programmable Device Category	Equivalent Gate Range
SPLD	Up to 500
CPLD	Up to 50,000
FPGA	Up to 5,000,000
ASIC	Up to 50,000,000

Table 2-1 shows the application space of each evolutionary step. For comparison purposes, Application Specific Integrated Circuits (ASIC) that are semi-custom, mask-programmed devices are included. As programmable logic densities have increased, ASIC densities have as well. But the lower density range of the ASIC market has been quickly won over by PLDs. ASICs have typically been relegated to very high density, very high speed applications. With time, PLDs have been closing the density and speed gap with ASICs.

2.1 Simple & Complex Programmable Logic Devices

Simple Programmable Logic Devices (SPLD), also known as Programmable Array Logic (PAL), is now an insignificant, rapidly shrinking part of the six billion dollar programmable logic market. It is, however, still of interest to examine their architecture since it laid the groundwork for the architecture of Complex Programmable Logic Devices (CPLD).

Typically, an SPLD consisted of a large switch network that allowed for programmable connections between device inputs and wide input AND gates. As pointed out in Figure 2-1, the inputs went into a product term array that served the purpose of logically ANDing signals together. The outputs of the product term array were then ORed together, creating an AND-OR plane of logic. The output of each large AND gate drove the data input of a flip-flop. Other routing choices were available for each device pin. For instance, some pins could be programmed as outputs and some pin signals could be used a flip-flop clock signals.

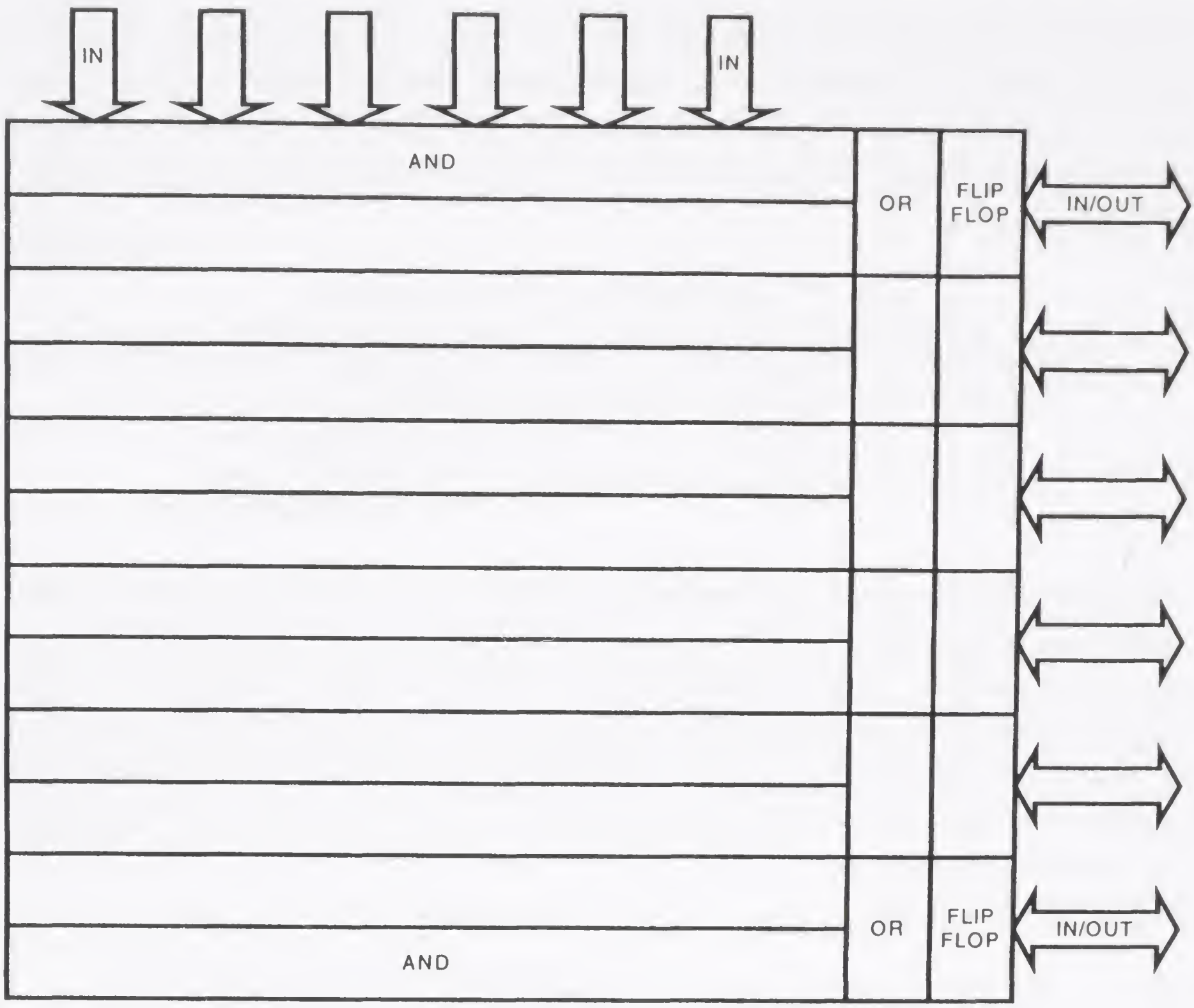

Figure 2-1. Block Diagram of a Typical SPLD

The most popular SPLD device was the 22V10. The name stemmed from the number of available pins on the device (22 being the total number of user available pins on the device, 10 programmable IO and 12 inputs) and the number of registers on the device (10). Many variations on this basic device were made. The general architecture remained the same and the number of IO pins and flip-flops varied.

SPLDs like their descendants, CPLDs, featured deterministic and fast pad-to-pad timing. Most SPLDs, however, had only one clock signal available in each device, one output enable and rather limited routing. For these reasons, SPLD's use was limited to implementation of small state machines, address decoders and to consolidate random glue logic.

As density and complexity demands increased, the SPLD architecture was no longer applicable. The first variations on this architecture were known as complex programmable logic devices (CPLD).

In 2003, CPLDs made up about 35% of the programmable logic market. These devices inherited the AND-OR structure from PALs, but offer more inputs and outputs and better sharing of product terms and more clock inputs.

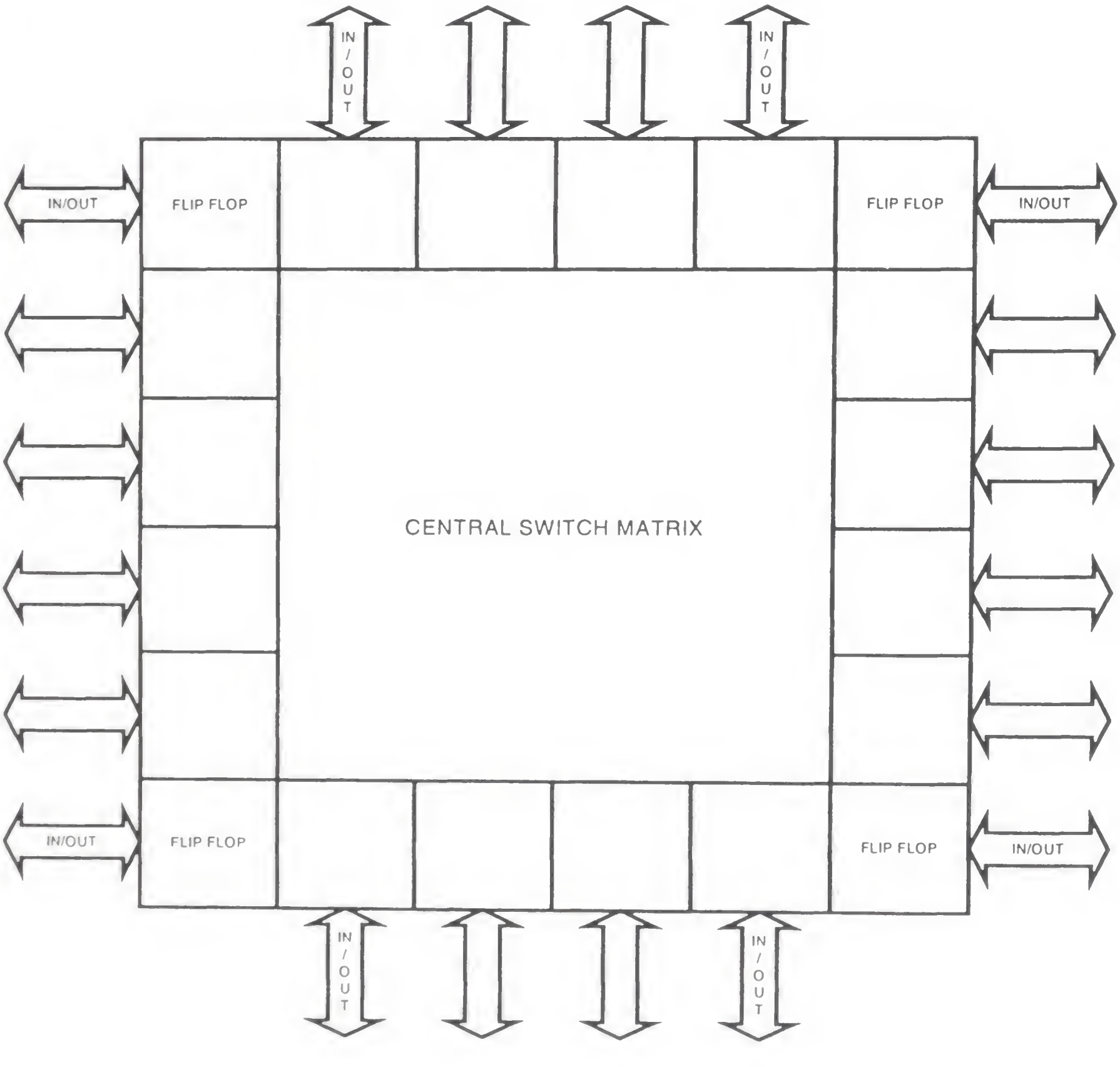

Figure 2-2. Block Diagram of a Typical CPLD

CPLDs have an architecture that results in deterministically calculable speeds with fast pad-to-pad delays. Like the SPLD, there are centralized routing resources. This typically takes the form of either a fully or partially populated central switch matrix (CSM). Looking at Figure 2-2, you will see that in many ways, a CPLD is similar to multiple SPLDs connected by the CSM. Since all signals are routed through the CSM, the CPLD has predictable timing.

The CSM also allows connection of IOs to flip-flops in the CPLD or to other IOs. Typically, each flip-flop has both set and reset inputs whose controlling signal can be flexibly assigned. The implementation logic of the CPLD is typically arranged in macrocells. Each macrocell consists of a wide input programmable logic gate (typically a NAND gate with an invertible output) and a flip-flop with set and reset controls. All paths through the macrocell are programmable and invertible. Each macrocell can therefore be a portion of a random logic function or a portion of a registered state machine.

Typical CPLDs contain anywhere from about 30 to 500 registers. These devices are typically used to realize wide input functions, state machines or data interface logic that is not register intensive. CPLDs are typically nonvolatile devices (meaning that they remember their configuration after power is removed).

The three basic characteristics resulting from the CPLD architecture are as follows:

- High speed
- Nonvolatile configuration
- Deterministic timing

These characteristics represent both the strength and weakness of CPLDs. Their high speed allows them to perform as well as a custom or semi-custom solution. The small board area they consume since they integrate discrete functions and do not need a separate memory to store their configuration helps reduce system cost. Finally, the deterministic timing makes it easy to design them into systems and to predict system performance. Unfortunately, these are strengths that designers view as essential features and thus these device requirements work against increasing CPLD densities using foreseeable process technologies. CPLDs often have high static power consumption, caused by their wired-OR interconnect structure with many sense amplifiers. The Lattice Semiconductor ispMACH 4000 and the Xilinx CoolRunner and CoolRunner2 families of CPLDs offer ultra-low static power consumption.

Since the basic CPLD architecture cannot expand easily to large arrays, CPLDs are inherently limited in size and offer few flip-flops. The limited size (and therefore the limited logical complexity of designable applications) aids in making CPLD design software simple and easy to use, providing rapid design compilation times.

2.1.1 Altera CPLD Architectures

The Altera Multiple Array MatriX (MAX) architecture is a typical CPLD architecture. This architecture represents a hierarchical arrangement of erasable Programmable Array Logic blocks using a two-dimensional array structure. It is pictured in Figure 2-3. The design provides multiple level logic, uses a programmable routing structure and is user reprogrammable based on EEPROM technology.

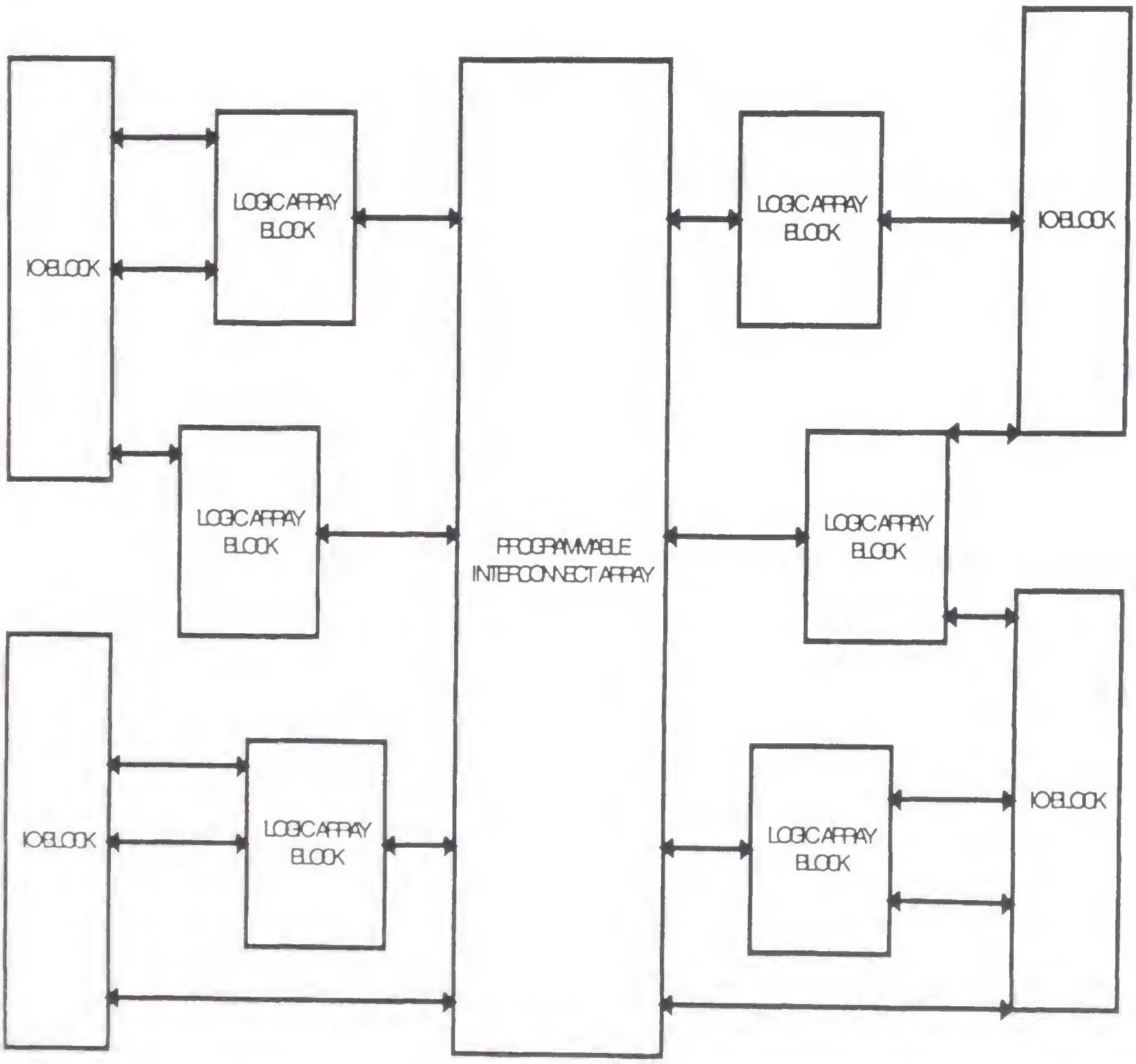

Figure 2-3. Block Diagram of the Altera MAX Device

The MAX 5000 series and the second-generation MAX 7000 series architectures consist of an array of large programmable blocks called Logic Array Blocks (LABs). Each LAB in the MAX 7000 family comprises 16 macrocells. Each macrocell in turn has a programmable-AND/ fixed-OR array and a configurable register. Thus, each macrocell represents a small PLD with five programmable product terms, and it can be configured for either sequential or combinatorial operation. Complex logic functions can be

formed using multiple macrocells. In addition, the Altera LAB architecture provides both sharable and parallel expander product terms ("expanders") that can be used to deliver more product terms directly to any macrocell in the same LAB. Finally, at the top level of the design hierarchy, signals are routed between LABs by a Programmable Interconnect Array (PIA). This global routing resource connects any signal source to any destination on the chip.

The MAX 9000 family uses EEPROM nonvolatile programming, and a logic hierarchy built from macrocells that are grouped into LABs as in the MAX 7000 family. However, the routing architecture of the MAX 9000 family uses the FastTrack technology. There are 96 routing channels in each row and 48 routing channels in each column.

2.1.2 Lattice Semiconductor CPLD Architectures

The Lattice Semiconductor MACH, pLSI and ispLSI families are variations on the MAX theme.

The MACH family is a collection of PAL-like blocks arranged around a central switch matrix for interconnect. A block diagram of the architecture is supplied in Figure 2-4. The number of PAL blocks is increased to increase the gate density.

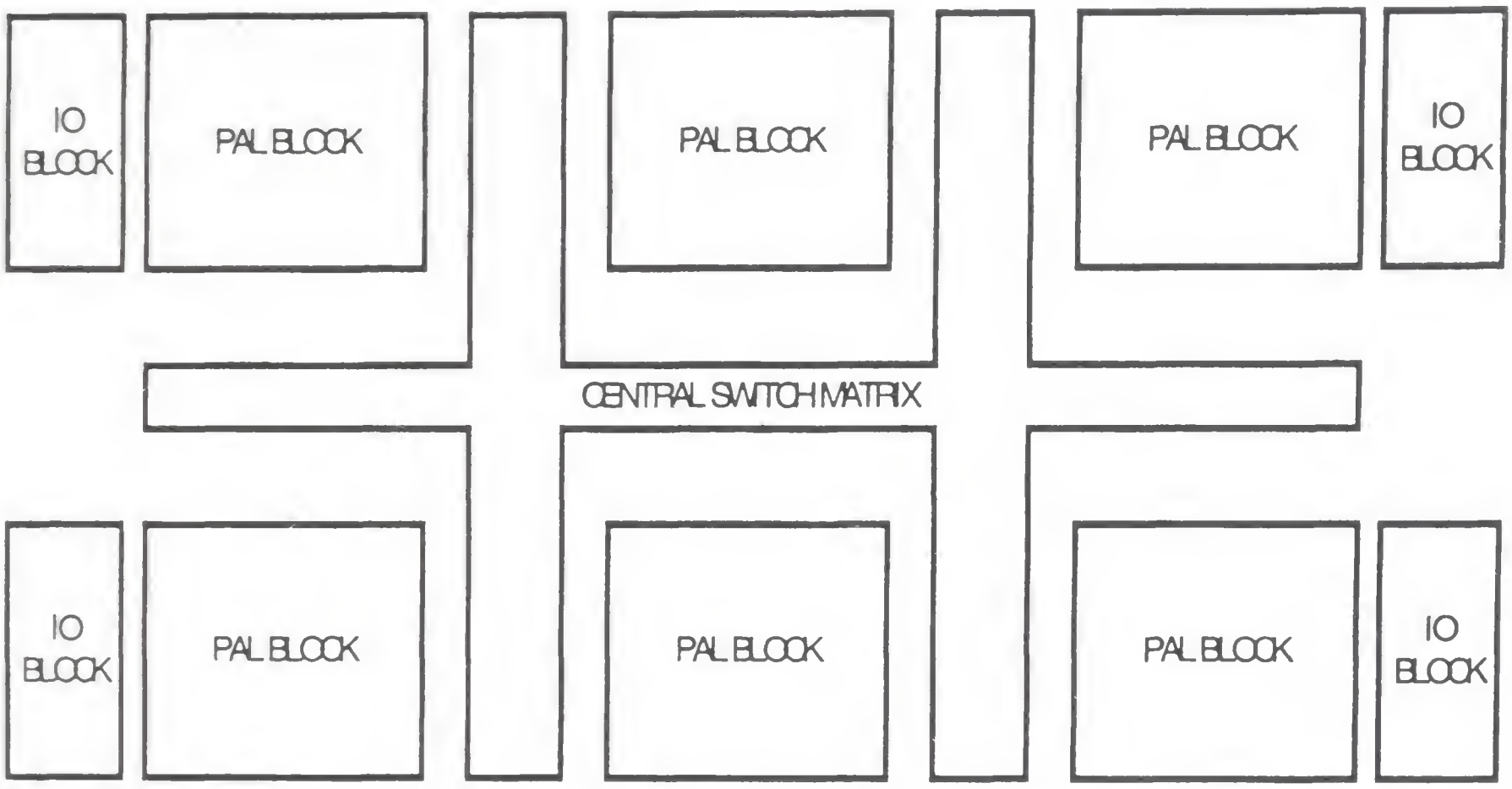

Figure 2-4. Lattice Semiconductor MACH Device Block Diagram

The pLSI and ispLSI devices have a ring of Generic Logic Blocks (basically PALs) around a switch matrix called a Global Routing Pool. As with the MACH devices, all interconnects pass through the GRP yielding predictable timing.

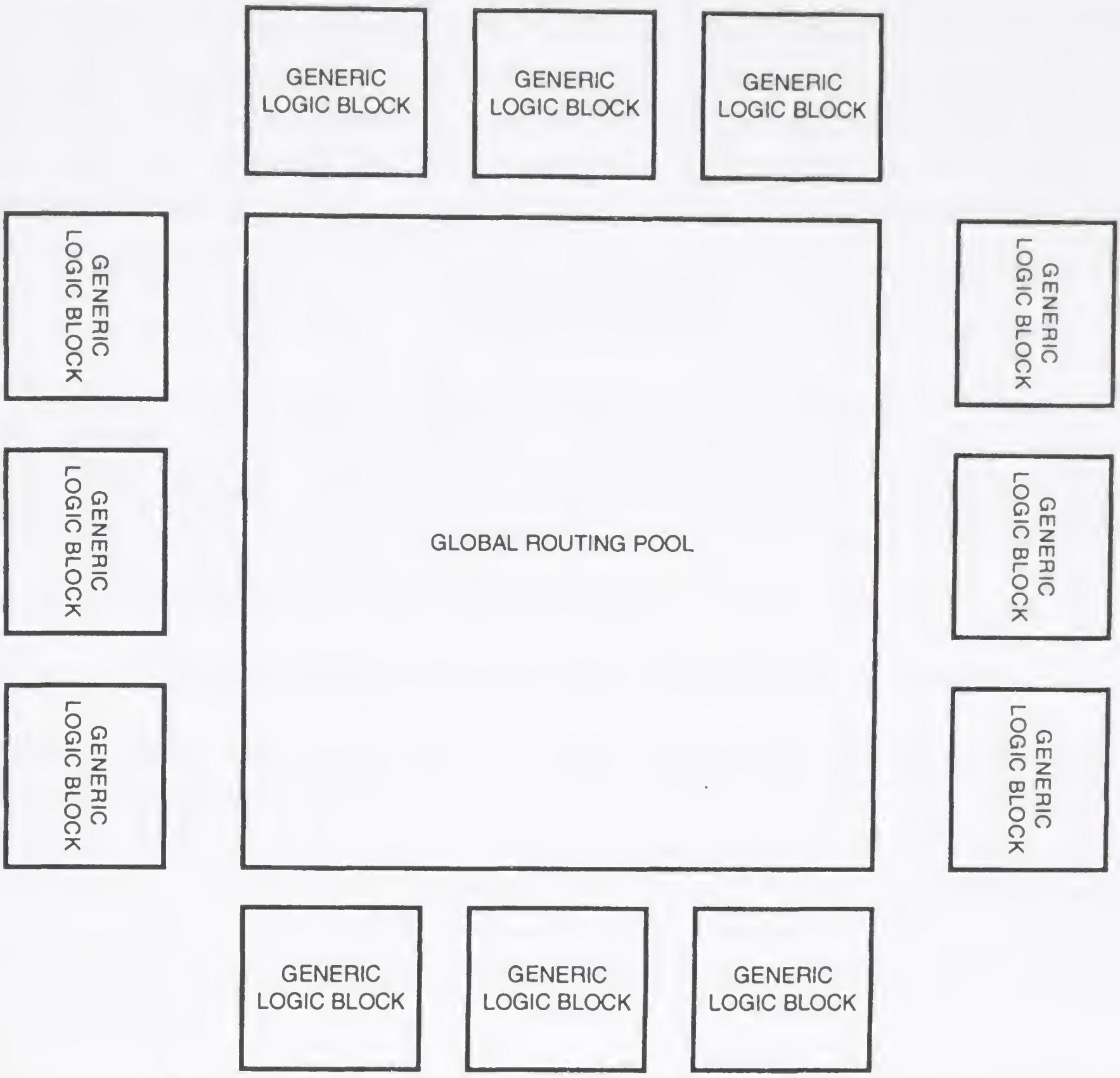

Figure 2-5. Lattice Semiconductor pLSI and ispLSI Device Block Diagram

2.1.3 Xilinx CPLD Architectures

The Xilinx 9500 family of CPLDs (including the 9500, 9500XL and 9500XV devices), as well as the Xilinx CoolRunner and CoolRunner2 are all subtle variations on what we have already seen. A collection of logic array blocks or optimized PALs programmatically connectable to one another through a switch matrix.

With the 9500, the switch matrix was unique in that it was a fully populated crossbar switch. This provided guaranteed routability since all paths were possible. For the next generation XC9500XL and XC9500XV devices, the switch was optimized and not fully populated.

The CoolRunner devices use an architecture similar to the Altera MAX but feature an exceptionally low power profile.

2.2 Field Programmable Gate Arrays

Field programmable gate arrays (FPGA) represent the most popular of the programmable device architectures. FPGAs made up about 55% of the programmable logic market in 2003. They have a more ASIC-like architecture with many flip-flops and distributed routing.

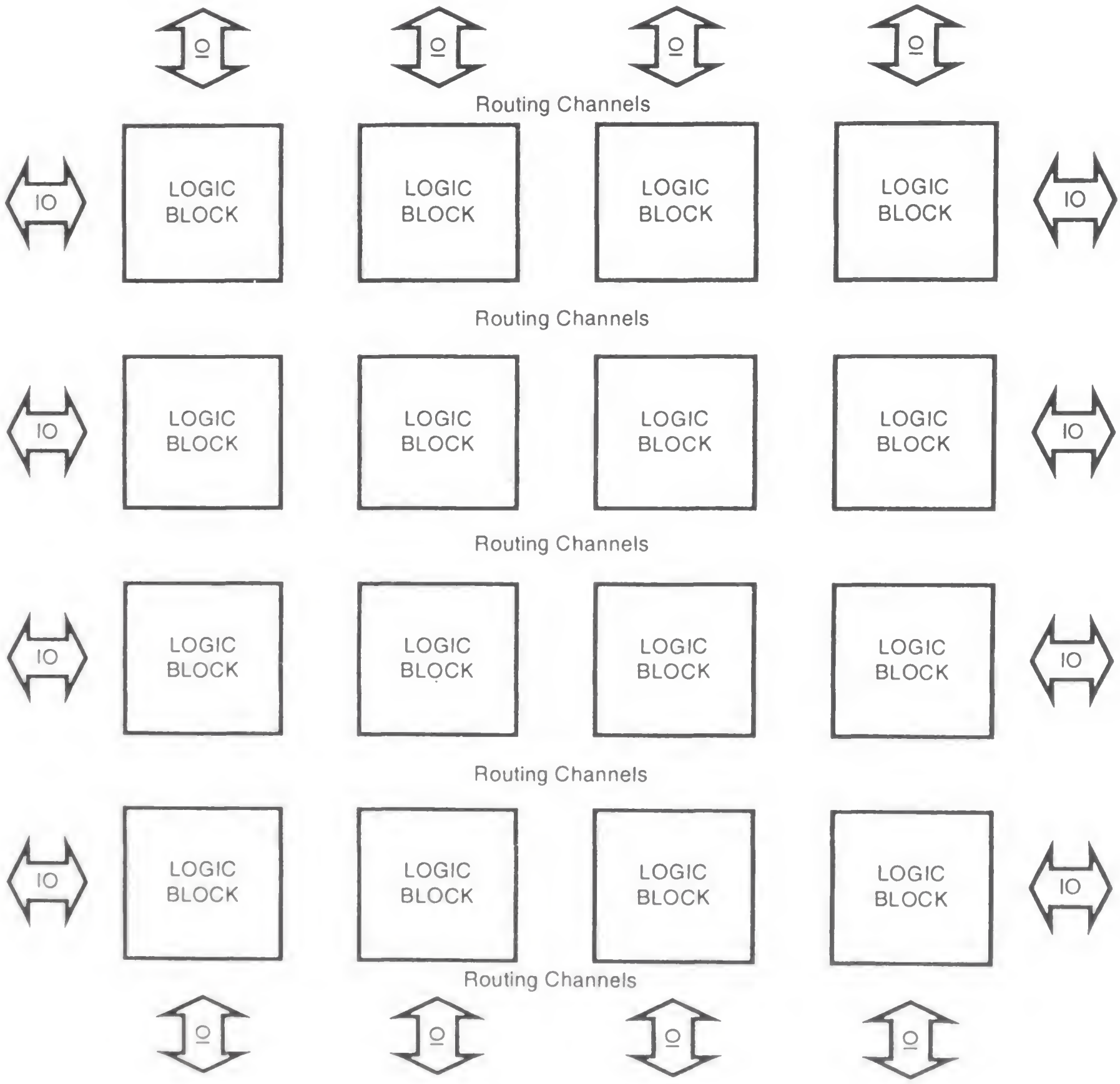

Figure 2-6. Generic FPGA Architecture

The basic FPGA architecture is shown in Figure 2-6. Although there are several varieties of FPGA architectures, they use the same basic approach. The variation is in the number and type of routing resources provided, the functionality of the logic block and the availability of prefabricated cores of specialized functionality. The specialized cores may include

microprocessors, high-speed communications transceivers, digital signal processors and other similar complex functions.

2.2.1 Xilinx FPGA Architectures

The Xilinx XC4000 family of devices typifies the FPGA architecture. These devices have a routing structure that allows arbitrary point-to-point routing but with limited routing resources. A design is realized by routing signals between configurable logic blocks (CLB). Each CLB consists of two four input look-up tables (LUT) that can act independently or have their outputs routed to one or two constituent latches or flip-flops. Other functions also available are the ability to access the configuration SRAM as a register and the ability to direct the look-up table outputs through another look-up table with an external signal to create functions of up to nine inputs.

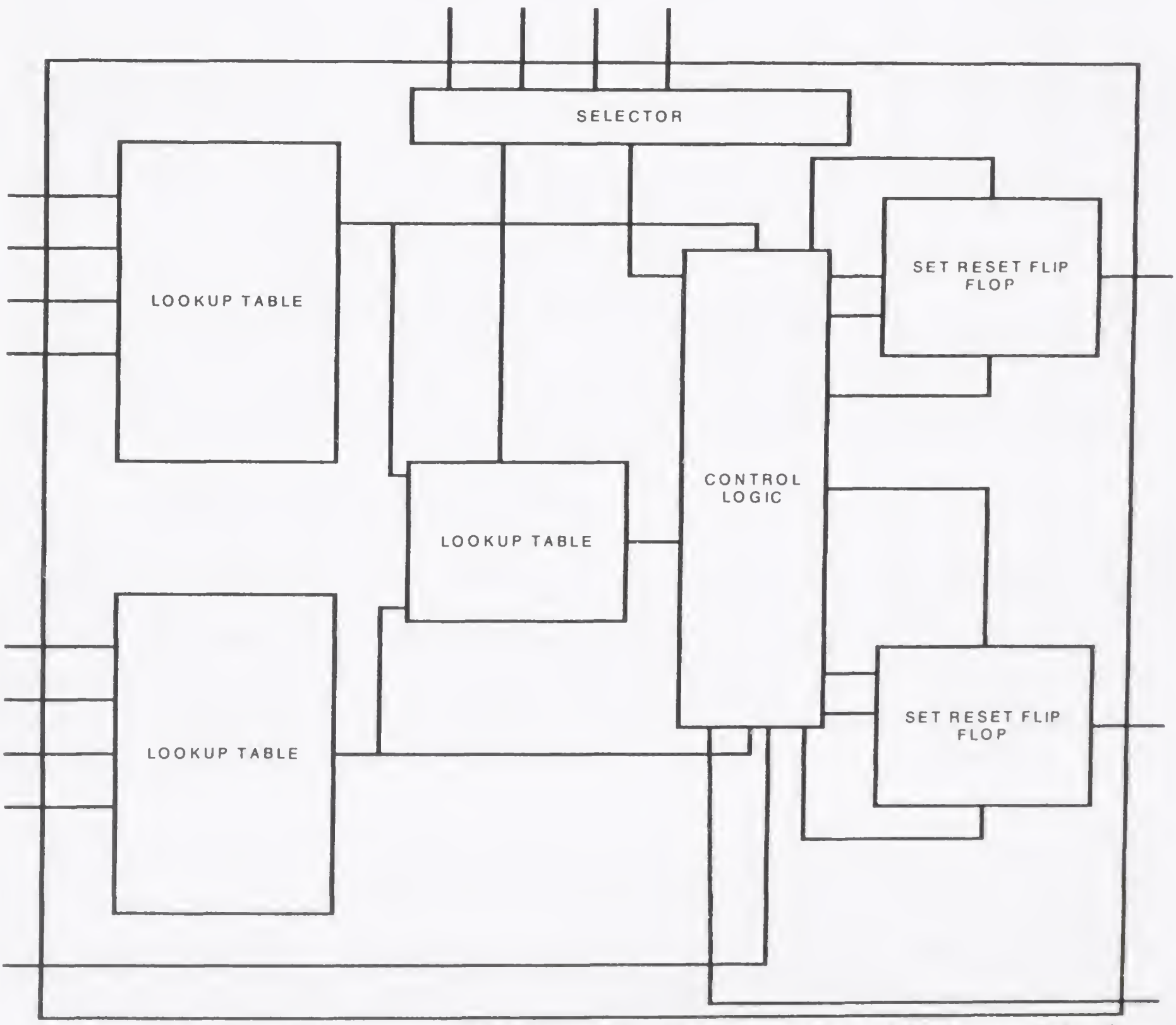

Figure 2-7. Block Diagram of Xilinx XC4000 Configurable Logic Block

The Configurable Logic Blocks (CLBs) are organized in a two-dimensional array separated by horizontal and vertical wiring channels. Each

CLB contains flip-flops, multiplexers, and a combinatorial function block that works as an SRAM based table look-up. Turning on pass transistors customizes connections between CLBs. The pass transistors selectively connect the CLBs to the interconnection resources, or interconnect lines between the horizontal and vertical wiring channels. SRAM cells that are scattered around the chip hold the state of the interconnect switches. Surrounding the CLB array and interconnect channels are the programmable IO blocks which connect to the package pins.

The overall architecture has powerful functional blocks connected to one another by versatile interconnect. This directs design software to pack as much functionality as possible locally in CLBs. In addition, the software tries to limit interconnect dependencies.

The design of the Xilinx CLB and routing architecture is slightly different for each product family. The first generation family (known as the XC2000) is no longer available. It is however interesting to understand its architecture. It contained a CLB with a single D flip-flop and a look-up table that can form any Boolean function of four variables, or two functions of three variables. The routing architecture used three resource types: direct connection, general purpose interconnects, and long lines. Direct connection lines were used to interconnect a CLB with bordering CLBs or IO blocks above, below, or to the right. General purpose interconnects were used for connections which span more than one CLB. There were four horizontal and five vertical general purpose interconnect lines between the array rows and columns, respectively. Each segment ran only the length of a CLB, and then entered a switch matrix that provided programmable connections to adjoining row or column general purpose interconnects. Finally, each horizontal wiring channel had one long line and each vertical wiring channel had two long lines that spans the entire array. These long lines bypassed the switch matrices. They route global signals (for example, clocks), or other signals that needed minimum skew at multiple fan-out points.

The second-generation family was known as the XC3000. In the XC3000 architecture, the logic block (CLB) is expanded and extra routing resources are provided. The CLB can fulfill any Boolean function of five variables or two functions of four variables. Two D-type flip-flops are provided to capture both cell outputs if needed. The routing architecture is similar to the XC2000 family except that each resource type has been improved. Direct connections are allowed to all nearest neighbors and an extra wiring segment is added to the horizontal general purpose interconnect.

As well, an extra long line is added to both the horizontal and vertical channels.

Compared with its predecessors, the XC4000 family adds evolutionary improvements to the basic Xilinx architecture. Greater logic capacity in each CLB is achieved using a two-level look-up table. The 13 input and four output CLB can form any of the following combinatorial logic functions:

- Two independent functions of up to four variables
- Any single function of five variables
- Any function of four variables with some functions of five variables
- Some functions of up to 9 variables

Compared with earlier families, the routing resources of the XC4000 family were significantly increased. The number of globally distributed signals has increased from two to eight, and there are twice as many horizontal and vertical long lines. The number of wiring segments has also more than doubled, and CLB connectivity is improved by allowing most CLB pins to connect to a high percentage of the wiring segments. However, the switch matrix connectivity was reduced to 50% of that of the XC3000 family. The increased efficiency of the associated place and route software indicated that changes in the routing resources were justified. It was demonstrated that FPGA connection blocks needed high flexibility to achieve a high percentage of routing completion, and that relatively low flexibility is needed in the switch blocks.

A significant variation from the XC4000 was the XC5000 device. Architecturally it is still a symmetrical array with SRAM based programmable logic and interconnections. The internal chip organization was dramatically changed. However, the device preserved pin-for-pin compatibility with and had an identical programming and control interface to the XC4000 family.

The logic blocks and their local routing connections were combined into a larger entity called a VersaBlock. The VersaBlocks provided logic and connectivity for efficient assembly of local logic functions. These local functions are then globally interconnected through a General Routing Matrix (GRM). This architecture provides five levels of interconnect hierarchy. This was to be used to exploit the locality of logic in typical digital designs efficiently.

Heavily interconnected logic macro-functions placed in bordering CLBs can be locally connected within the VersaBlock. This allows the GRM resources to be devoted to connections between macro-function blocks.

The VersaBlock contains a CLB composed from four separate logic cells (called LC0 through LC3), with a local interconnect matrix. Note that each of the four logic cells within the XC5000 CLB is similar in structure to the original XC2000 family CLB; with a single D flip-flop and a four-variable Boolean function generator. However, grouping four of these independent logic cells in a tightly coupled VersaBlock unit allows efficient high-speed carry chains or high fan-in logic functions to be easily created.

The alternative to this approach is to make interconnect less versatile (and therefore less expensive). With this simplified interconnect, more resources can be dedicated to logic cells by increasing their number.

A device of this sort has a two-dimensional mesh array structure that resembles the gate array "sea of gates" architecture. Like the Xilinx architecture, Static RAM programming technology is used to specify the function performed by each logic cell and to control switching connections between cells. An example of this device is the Xilinx XC6200 family of FPGAs. The design contains 1024 identical logic cells arranged in a 32 X 32 matrix. The design is considered to be a mesh-connected architecture since each cell is directly connected to its nearest north, south, east, and west neighbors. As well as these direct connects, two global interconnect signals are routed to each cell to deliver clock and other "low skew requirement" control signals. The basic array architecture incorporates both nearest neighbor and global connections in the logic cells. Besides these logical connections, row select lines and bit select lines are connected to program each cell's SRAM bits.

The basic building block of the XC6200 design is a configurable cell containing multiplexers and a function unit. Multiplexers that select the source for the X1 and X2 inputs precede the function unit. The function unit can produce any logic function of the two inputs, or of acting as a D-type latch. There are four more multiplexers that select the function output or one of the external inputs for routing to each of the four outputs (north, south, east, and west).

A unique feature in the XC6200 IO pad design is its capacity to provide simultaneous input and output on the same pin when communicating with another device of the same family. This is done through a 2-level (ternary)

logic-signaling scheme in which IO pads sense whenever two outputs are driving each other by a contention scheme. Even during contention, the pad can deduce the correct input value and pass it along to the internal circuitry. This makes it easier to partition a single design across multiple FPGAs because the increased connectivity reduces pin limits on communications bandwidth.

The Virtex family of FPGAs (which includes the Virtex and VirtexE devices) represents the fourth generation architecture for Xilinx. It evolved from the XC4000. This architecture represents most devices currently being shipped by Xilinx in 2003. This architecture features more routing resources, a modified CLB and configurable block RAM. A ring of routing resources surrounds the implementation that simplifies interconnections among the LUTs, flip-flops, and GRM. Extra global routing resources are made available and 2 high-speed pass-through routes are included in each CLB.

The most recent addition to the Xilinx family of FPGAs is the Virtex2 family (which includes the Virtex2 and Virtex2Pro devices). Once again, this architecture features further improvements to the CLB and a wider variety of routing resources to promote faster design implementations. In addition, the Virtex2 has more configurable block RAM as well as specific architectural features to simplify clock management and a support wider variety of IO standards.

The Virtex2Pro devices add PowerPC processors and programmable gigabit speed IO transceivers to the fabric of the FPGAs. This allows unprecedented capacity to receive, process and transmit data within the physical boundaries of a single programmable device.

2.2.2 Actel FPGA Architectures

In the Actel ACT™ family FPGAs, a logic module matrix is arranged as rows of cells separated by horizontal wiring channels. This organization is similar to that found in the traditional style of Mask Programmed Gate Arrays (MPGAs). Vertical interconnect segments of varying lengths are available. Vertical segments in input tracks are permanently connected to logic module inputs, and vertical segments in output tracks are permanently connected to logic module outputs. Long vertical segments are available which are uncommitted and can be assigned during routing. The horizontal wiring channel resources are also segmented into varying lengths. The minimum horizontal segment length is the width of a single logic module,

and the maximum horizontal segment length spans the full channel. Any segment that spans more than one-third of the row length is considered a "long horizontal segment". Connections between interconnect segments are permanently formed using the antifuse. Dedicated routing tracks are used for global clock distribution and for power and ground tie-off connections. Actel has three generations of FPGAs, called ACT1, ACT2, and ACT3.

In contrast to the Xilinx FPGA that uses a complex CLB cell, the Actel approach uses small and simple logic modules. This does not imply that the Actel design has inherent disadvantages compared with the Xilinx approach. Research has shown that both of these approaches have merit.

Research results suggest the best choice for a programmable block depends on the speed performance and the area requirements of the routing architecture. The low-impedance and small area Actel antifuse structure is better suited for use with a simple logic module. On the other hand, the larger area and higher resistance Xilinx SRAM controlled transistor switch is more apt for a complex logic cell.

The ACT1 family Logic Module (LM) is an 8-input, one output function which can be used to build the four primitive logic functions (AND, OR, NAND, NOR) with two through four inputs. The basic ACT1 Logic Module circuit uses multiplexers to create programmable logic functions. The LMs can also be used to make latches, flip-flops, XORs, AND-ORs and other logic structures. Actel does not include dedicated hardwired latches or flip-flops in the ACT1 array since they can be built from LMs wherever needed in the design. The ACT1 family uses 22 metal signal wiring tracks in each horizontal channel and 13 vertical tracks that lay on top of each column of LMs.

The ACT2 family is Actel's second generation of FPGAs. It uses two different types of logic modules: a Combinational (C) Module and a Sequential (S) Module. The C module with eight inputs and one output is similar in functionality to the LM used in the ACT1 family. The S-Module is designed to set up high-speed D flip-flops or latches within a single cell efficiently.

An S-module can create an up to 7-input Boolean function followed by a D-type flip-flop or a latch. The S-Module can also be configured with a transparent latch. Then, like the C-Module it can also carry out a purely combinatorial 8-input function. C-Module and S-Modules are paired and then grouped in alternating pairs to form the rows of the ACT2 array. The

ACT2 routing structure is also similar to that of ACT-1, with the same three types of routing resources:

- Vertical input and output segments
- Clock tracks
- Horizontal wiring tracks

However, there are 14 extra tracks in each horizontal wiring channel and two additional tracks in each vertical column.

ACT3 is Actel's third generation FPGA family that uses the same basic array architecture with improved versions of the ACT2 family logic modules. The new C-Module is functionally equivalent to that of ACT2, while the S-Module has been expanded to include a full C-Module driving a flip-flop. The ACT3 architecture contains four clock networks. Two of which are dedicated high-performance clock networks, and two are general-purpose networks. The ACT3 architecture continues to use the routing resource structure of the ACT2 design with horizontal wiring channels and vertical wiring tracks that overlay the logic modules.

2.2.3 Altera FPGA Architectures

The FLEX 8000 series was Altera's first PLD based on SRAM programming technology. This series used a fine-grain hierarchical architecture including 4-input look-up table Logic Elements (LE) as the basic functional building block. LEs are grouped into sets of eight to create LABs as in the earlier family designs. These blocks are arranged into rows and columns. Connections between LEs are provided by horizontal and vertical FastTrack interconnect channels that span the chip. Both the Logic Elements and the FastTrack interconnects are SRAM programmed in a similar fashion to the Xilinx technology discussed earlier.

The FastTrack interconnect technology is used in the FLEX 8000 part. The LABs are arranged into a two-dimensional array separated by horizontal and vertical FastTrack wiring channels that span the entire array. An advantage of this device wide routing is that it provides predictable wiring delays when compared with segmented FPGA wiring schemes which use a variable number of programmable interconnection points in the routing path. The FLEX 8000 family parts have either 168 or 216 routing channels in each row and 16 routing channels in each column.

Each column of LABs has dedicated lines that route signals out of the LABs and into the FastTrack column. The column interconnects can then drive IO pins or feed into the row interconnects to drive other LABs. The number of wiring channel routing resources varies by family and part type.

Each row of LABs has a dedicated row interconnect for routing macrocell inputs and outputs. The row interconnects can then drive IO pins or feed other LABs on the chip. Each macrocell in the LAB can drive up to three separate column interconnect channels. A row interconnect channel can be fed by the output of a macrocell through a 4-to-1 multiplexer that it shares with three column channels. If the 4-to-1 multiplexer is used for a macrocell-to-row connection, then the three column signals can access another row channel by an extra 2-to-1 multiplexer.

Recent Altera FPGA architectures, like the Stratix and Stratix GX have been subtle variations on the Xilinx Virtex II and Virtex II Pro approaches. Architecturally similar, these devices included different processors than the Virtex II Pro, different block RAM and high-speed transceiver resources.

Chapter 3

IN-SYSTEM CONFIGURATION TECHNOLOGIES

1. Introduction

In the previous section, we looked at the architectures of programmable logic devices in general. Architecture considerations are one of the primary reasons in determining which programmable device a designer will choose. Other considerations are, of course, price, availability, implementation tool performance and, often, corporate guidelines.

However, in designing a reconfigurable system the reconfiguration technology becomes another consideration. In this section, we will examine the variety of configuration technologies used in programmable devices.

The devices discussed so far fall into two broad configuration families: volatile and nonvolatile devices.

Nonvolatile devices keep their configuration information even when the device is powered off. Typically, SPLDs and CPLDs are nonvolatile. This means the boot-up time for these devices is instantaneous. When powered up, a system made with these devices (if they are previously configured) is ready to go.

Volatile devices forget their configuration after power down. This means that these devices need to be reminded of their configuration at power on. This is usually accomplished by keeping the configuration information in a nonvolatile store like a PROM or a disk. The implication is that volatile devices need some finite (and measurable) amount of time after power on to be reloaded with their configuration before being ready to go. FPGA devices typically are volatile.

Table 3-1. Configuration Technology Characteristics

Feature	Antifuse	SRAM	EEPROM	FLASH
Nonvolatile	Yes	No	Yes	Yes
Reconfigurability	No	Yes	Yes	Yes
Endurance	1 cycle	Infinite	< 10,000 cycles	~ 100,000 cycles
Programming time	Minutes	< 1 second	< 10 seconds	< 2 minutes
External Prom	No	Yes	No	No
Power-up time	Instant	< 1 second	Instant	Instant

A small subgroup of FPGAs uses antifuses to control their interconnect structure. Thus, these devices preserve their configuration when powered down, they power-on instantly, and they need no external configuration memory. This non-volatility comes at the cost of reconfigurability. Antifuse-based FPGAs are usually one-time-programmable. In addition, device programming often takes several minutes. These antifuse-based devices represent about 6% of the FPGA market.

2. Nonvolatile Configuration Technologies

We will examine three separate configuration technologies in this section:

1. Antifuse cells
2. Electrically erasable and programmable cells
3. Flash erasable and programmable cells

2.1 Antifuse Cells

Actel, QuickLogic and others have introduced commercial products that use antifuse programming. In Actel FPGAs, a Programmable Low-Impedance Circuit Element (PLICE) antifuse element is used. The normally high antifuse resistance (>100 Megaohms) is permanently changed to a low resistance (200-500 ohms) by applying suitable programming voltages. The programmed antifuse is used to make a direct electrical connection between two metal lines. Adding three specialized masks to a standard CMOS process is needed to make the PLICE antifuse. The physical structure illustrated in Figure 3-1, consists of an Oxide-Nitride-Oxide dielectric layer sandwiched between a top polysilicon layer and a bottom N+ diffusion layer. Applying a high voltage (about 18V) across the device and driving a high

current through the link dielectric completes the programming. This causes the dielectric to melt and results in a conductive link between the top and bottom terminals.

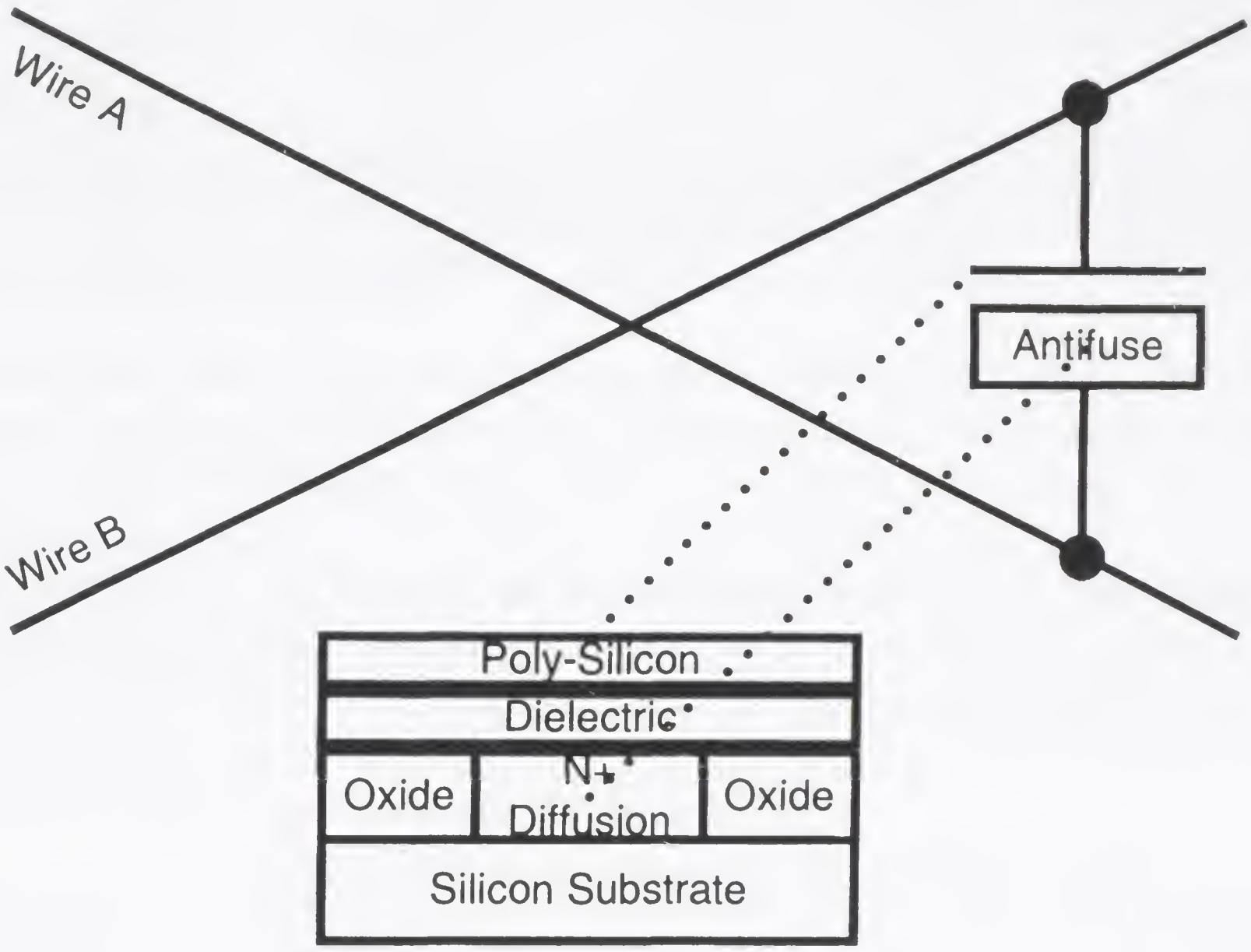

Figure 3-1. The Anti-Fuse Element

QuickLogic also adds a unique three-layer structure to the standard CMOS process to create their antifuse element, that they call a ViaLink. The ViaLink uses an amorphous silicon layer that is sandwiched between the first and second metal layers. An unprogrammed ViaLink has greater than 1 Mohms resistance and, like the PLICE antifuse, is programmed by applying a higher than normal voltage. The resulting high current through the amorphous layer causes it to permanently change to a conductive state with a typical resistance of only 80 ohms. The area occupied by these antifuse elements is small when compared to the other programming alternatives. While this contributes to improved on-chip gate density, the large area needed for the high-voltage transistors needed to support programming offsets it. Another disadvantage of the antifuse technologies is that they need adjustments to the standard CMOS process.

Since antifuse technology physically alters the connections irreversibly, the approach does not lend itself to use in reconfigurable systems.

2.2 Electrically Erasable and Programmable Cells

EEPROM technology was the first electrically erasable technology used for CPLDs. The programmable element is a special thin oxide capacitor that conducts a small current when enough voltage is applied across the oxide. The tunnel oxide, roughly 80 Angstroms thick, is used to inject or extract charge from a floating gate by Fowler-Nordheim (FN) tunneling. The floating gate is connected to the gate of a sense transistor in order to determine the programming state. Besides the tunnel oxide capacitor and sense transistor, two more transistors and an added control capacitor are needed to create a single EEPROM cell that can be programmed and erased.

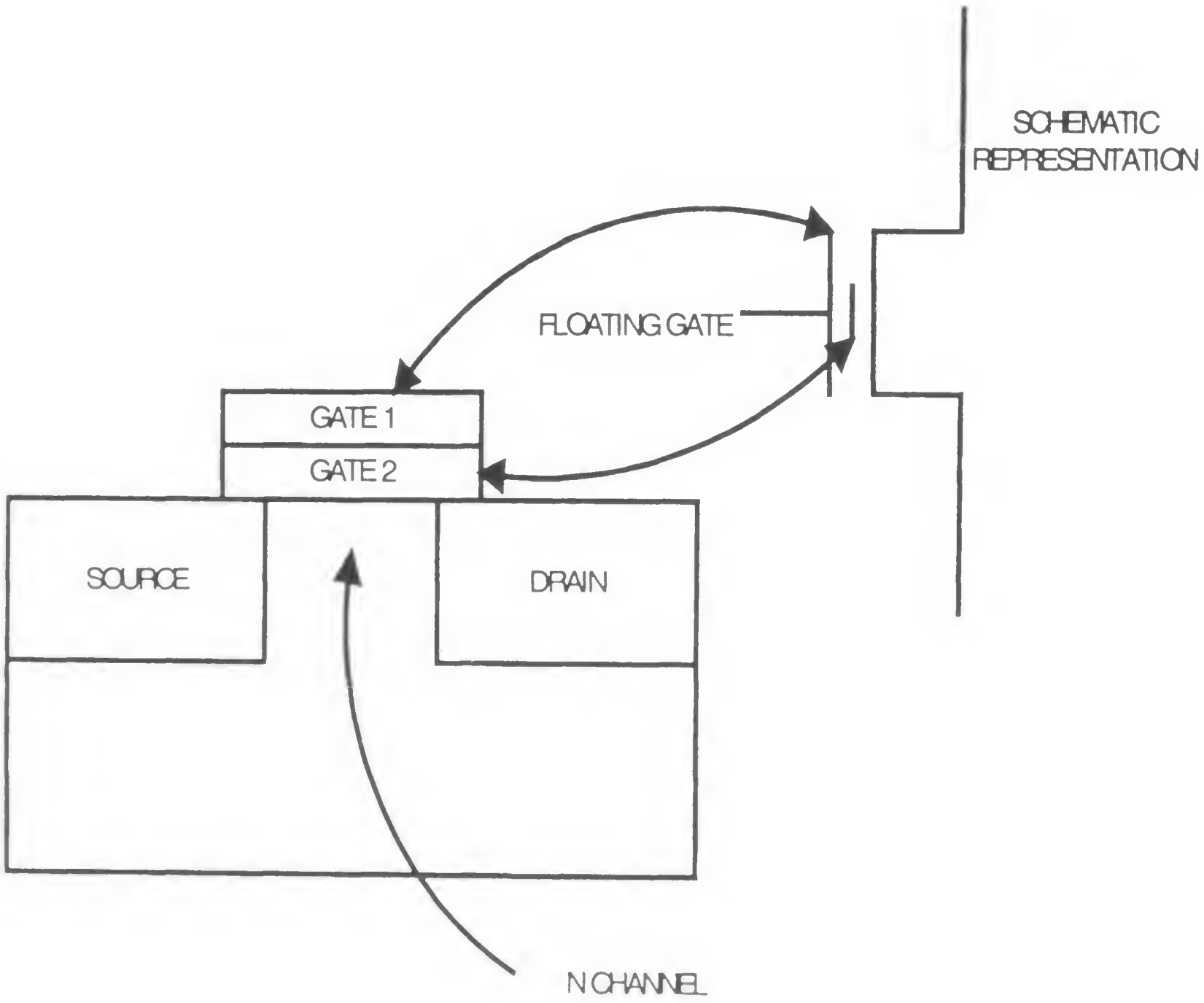

Figure 3-2. Typical EEPROM Cell

Specifically, an EEPROM cell is a MOS transistor that stores charge on an electrically isolated, conductive capacitor plate called a floating gate. A typical cell is depicted in Figure 3-2. The floating gate is located above the transistor channel. The charge on the floating gate produces an electric field

that changes the conductivity of the channel. The measure of the channel's conductivity matches the amount of the analog value stored.

A typical N-channel EEPROM cell includes an N-channel silicon-gate storage transistor. This transistor uses a floating first layer polysilicon gate (floating gate), directly accessed by a second stacked polysilicon gate (control gate), to trap and store electrons for long periods (typically decades).

The N-channel EEPROM cell is considered to be in an erased state when the floating gate has a net negative charge because of the presence of "hot electrons" injected from the drain. When the cell is in a programmed state, the electrons on the floating gate keep the N-channel transistor in a logical off state. When the floating gate is strongly programmed to a positive charge, the floating gate transistor's channel becomes conductive. This state corresponds to a binary digit, such as a logic 1.

Conversely, the cell is considered to be in a programmed state when there are no electrons on the floating gate and thus no net negative charge on the gate. To erase the cell, the energy of the electrons stored on the floating gate is raised until the electrons can "tunnel" through the tunnel dielectric from the gate to the source. When the cell is erased, the N-channel transistor is in a logical on state. Note that N-channel EEPROM cells are preferred over P-channel EEPROM cells because of the programmability and speed advantages of N-channel EEPROM cells.

If the EEPROM cell is erased, the floating gate becomes strongly negative. In this case, the EEPROM cell is nonconductive, corresponding to the complementary binary digit, a logic 0. Either programming or erasing cells in an array makes digital nonvolatile memory.

When a high voltage (typically greater than 10V) is applied over the thin insulator, electrons travel to and from the floating. This mechanism programs the cells. Erasure of the cells is affected by reversing the voltage applied during the writing process. This technique is known as hot electron injection. The high voltages needed for programming are typically produced on-chip and derived from the device supply voltage.

The tunnel oxide capacitor transports charge to and from the floating gate, which controls the sense transistor. Two extra transistors are used for the program and read operations. A control gate capacitor transfers voltage to the floating node for program and erase operations. Compared with

standard CMOS logic processes, three more device structures are created for the EEPROM cell: the tunnel oxide capacitor, the control gate capacitor and the high-voltage transistor. The resulting process complexity makes the scalability of the process and the EEPROM cell more difficult in future technology generations.

2.3 Flash Erasable and Programmable Cells

Technically, flash technology is a variation on the EEPROM technology described above. The physics of cell programming and erasure is different from that of EEPROM cells.

Like EEPROM cells, high voltages are needed for programming and erasure. These high voltages (often greater than 10V) are typically produced on-chip and drawn from the supply voltage.

The flash EEPROM cell typically incorporates its floating gate into the device structure for improved cell area. By adding an NMOS transistor in series, the flash transistor can be incorporated into the basic cell.

The behavior of an individual flash transistor is changed with a program or an erase operation. When a flash transistor is in the erased state, the threshold voltage (that is, the voltage at which the device turns on) is about 1 V. During programming, the threshold voltage increases above 5.5 V, so the transistor does not turn on for a logic operation.

The physical implementation of the flash transistor includes a floating gate polysilicon layer that is isolated from the silicon substrate by a thin oxide layer roughly 100 angstroms thick. Above the floating gate is the control gate polysilicon layer, with an insulating oxide-nitride-oxide layer between them. The control gate is driven by internal logic circuits while the floating gate is unconnected. When the flash transistor is in the erased state, there is no net charge on the floating gate. By changing the electrical charge on the floating gate, the threshold voltage may be increased.

The structure of the flash memory cell and EEPROM memory cell is therefore similar.

During the programming operation, channel hot electrons (CHE) are created near the pinch-off region. Some CHEs have enough thermal energy to pass through the thin oxide and remain on the floating gate. The collected electrons create a net negative voltage on the floating gate that opposes the

electric field emanating from the control gate. The result is a net increase in the threshold voltage.

Applying 0 volts to the control gate and around 10 volts to the source erases the flash transistor with the drain left floating. The electric field between the floating gate and the source node is increased to the point where Fowler-Nordheim tunneling takes place. Excess electrons are transported from the floating gate to the source. The transistor is designed to make the erase process self-limiting. The electric field decreases as electrons are removed from the floating gate. FN tunneling effectively stops when the floating gate is electrically neutral.

Once the basic memory cell is in place, additional control logic must surround it to allow addressable reading and writing. The fully controlled and programmable flash cell is typically one transistor smaller than the equivalent EEPROM cell. That results in a simpler cell structure, smaller cell size and potentially higher integration density.

Flash EEPROMs typically use Fowler-Nordheim tunneling, as opposed to hot-electron injection, for cell programming as well as for cell erase. A voltage signal, usually less than 25 volts, is applied to the control gate. The control gate is capacitively coupled to the floating gate. The drain is held either at ground potential or at a voltage less than that applied to the control gate, and the source is held at ground potential. Under such conditions, Fowler-Nordheim tunneling occurs, in which electrons from the drain, tunnel through a thin layer of SiO2 (tunnel dielectric) to the floating gate.

A conventional EEPROM cell electrically induces Fowler-Nordheim electron tunneling to erase the floating polysilicon gate. A high voltage signal (typically greater than 10V) is applied to the cell drain while the control gate is held at ground potential and the source is left at a floating, or unspecified, voltage potential. As a result, the electrons stored on the floating gate will tunnel through the tunnel dielectric to the source.

A conventional EEPROM cell contains an extra "select" gate to control erasure of that cell. By providing a byte-decode transistor for each EEPROM cell in a memory array to control its select gate, selective erase of individual cells or bits in the array can be achieved.

Although selective erase can thus be achieved, the extra select gate, for example, causes an EEPROM cell to be larger. The flash EEPROM cell does not contain an extra select gate and thus is smaller than a conventional

EEPROM cell. However, a memory array of flash EEPROM cells typically cannot be selectively erased because of the absence of select gates.

It is usual for memory arrays that use flash EEPROM cells to employ a "chip-mode" program cycle. First, all the cells in the array are programmed (logic off state). Second, all the cells in the array are erased (logic on state). Lastly, individual cells in the array are selectively programmed, while other cells remain in the erased state. This improves cell endurance. This means that the cell can endure more erase and program cycles if this technique is used. Note that all the cells in the memory array are programmed first before they are erased to avoid "over-erasing". For an over-erased cell, unselected cells can become leaky leading to false sensing of a selected bit on the same bit line and it will also be difficult to program the bit again.

2.4 Volatile Configuration Technologies

In contrast to nonvolatile technologies, nonvolatile approaches require a static configuration memory store to be coupled with the configurable device. As you might guess from its name, a volatile device loses its configuration information when power is removed.

The advantage to this technology is the storage cell is smaller so greater design logic densities can be built on a single chip. An advantage as well as a disadvantage is the need for an external configuration store. It's a disadvantage since a separate device is needed which will increase the cost and board space. It's an advantage since sophisticated users can design an associated configuration memory system that allows sharing and use of low cost off-the-shelf memories. It also allows users greater control of the activation sequencing of the devices at power-up.

In the next section, we will look at a typical volatile memory cell.

2.4.1 SRAM Cells

The Static Random Access Memory (SRAM) FPGA programming technology that was first introduced by Xilinx is also used in designs by Altera, Lattice Semiconductor and others.

Programmable connections in these FPGAs are made using multiplexers, transmission gates, or pass transistors. The paths through these are controlled by information stored in their controlling SRAM cells. Since the static RAM is volatile, these FPGAs must be programmed to set the circuit configuration

each time that power is applied to the chip. This can be carried out automatically through a serial connection to an attached ROM (PROM) or controller. Another approach would be to connect it in parallel mode using an attached processor or controller that addresses the FPGA as a normal static RAM. The chip area needed by the SRAM logic and interconnect programming circuitry is the large. A typical SRAM cell uses from 4 to 6 transistors. More devices will be needed for the transmission gates or multiplexers of the surrounding decode logic. A basic six-transistor SRAM cell is depicted in Figure 3-3. The basic programming cell consists of cross-coupled inverters that store the programming value. The value stored in the cell can be changed using the input transistor. The input transistor can drive more strongly than the competing inverter so it can overpower the feedback inverter to change the state of the cell.

To write to the cell, the BIT line is driven to the needed logic value, say a logic 1. The NOT BIT line is driven to its complement (a logic zero in this case). The WORD line is then selected (driven to a logic 1) and the cross-coupled inverters store the logic 1 after the WORD line is deselected.

To read the cell, the BIT and NOT BIT lines are pre-charged to Vdd. Then the WORD line is selected and a set of sense amplifiers on the bit lines, compare the voltage difference between BIT and NOT BIT to determine the stored logic value

Since these devices are produced using standard CMOS SRAM fabrication techniques, they can immediately benefit from advances in SRAM CMOS processes.

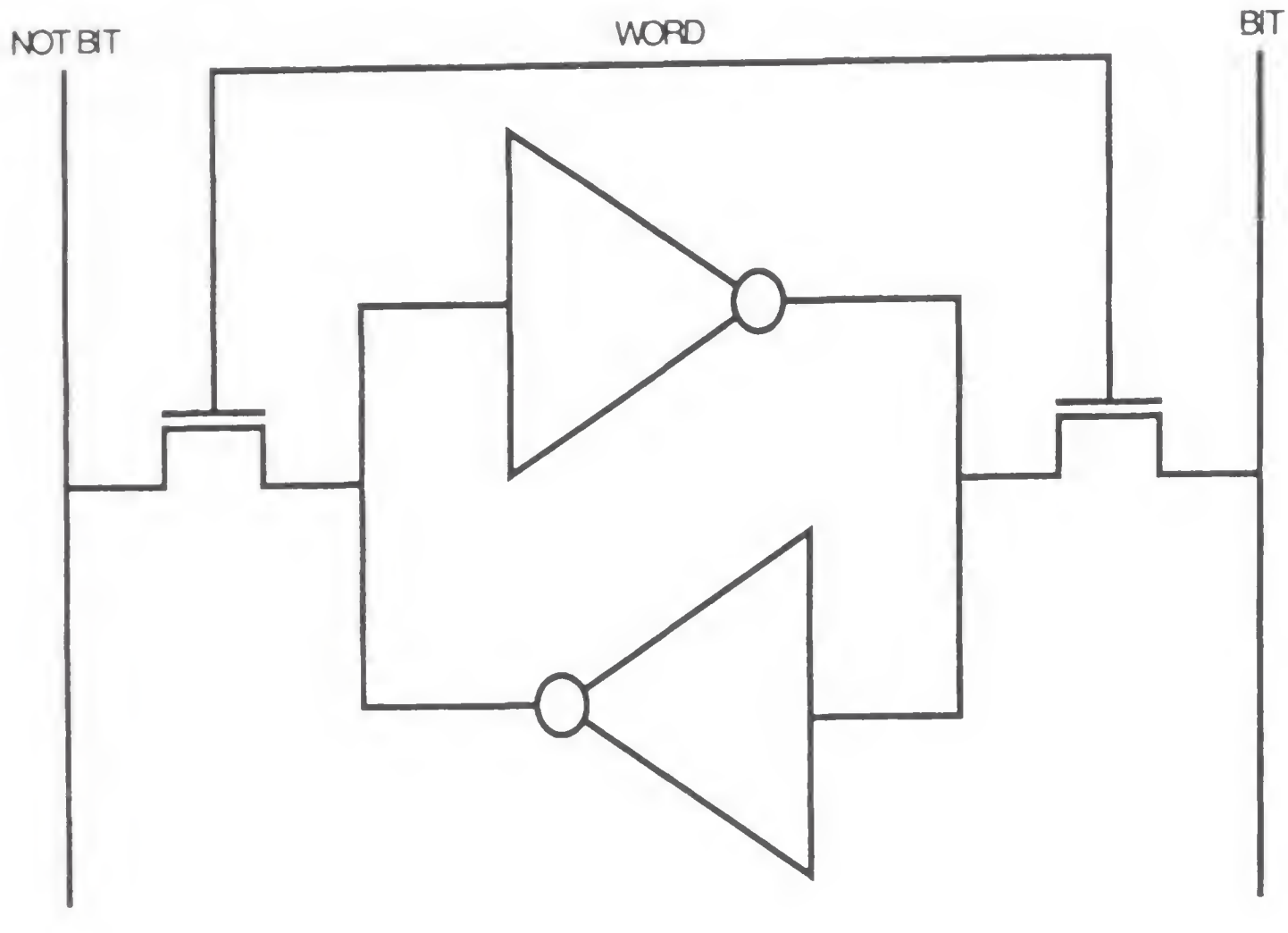

Figure 3-3. Basic 6 Transistor SRAM Cell

The volatile nature of the SRAM programming can be either a disadvantage or a major advantage. It imposes a system level overhead for ROM storage and power-on initialization time. On the other hand, programming nonvolatile devices may need stronger power supplies on board that would not be needed for SRAM devices. As well, reconfiguring SRAM devices is fast.

3. Configuration Access Ports

So far, we have seen there are several types of programmable logic architectures and that each has its advantages and disadvantages for each application. Volatile architectures may have advantages where reconfiguration cycles and configuration time are valued but a disadvantage if board space is at a premium. Nonvolatile architectures have definite advantages in those applications that need instantaneous power-up or have limited board space but may be at a disadvantage if the device is to be frequently reconfigured.

The missing piece is how to configure these devices. In this section, we will examine the access mechanisms that device manufacturers have made available to system designers to configure their PLDs. Often device manufacturers provide access by a multiplicity of mechanisms. One

approach that is almost universal across FPGAs is that a microprocessor can be used. Also common is the ability to use a PROM and have the FPGA control the configuration process itself.

These many possibilities allow the systems designer more freedom in selecting a suitable approach. If the system is sensitive to the time it takes to configure the devices and device pin resources are not strictly limited the parallel access approaches should be considered. In parallel approaches, multiple configuration data bits are transferred to the device with each controlling clock pulse through a parallel interface. If the interface is N bits wide then N bits can be programmed at a time.

If, on the other hand, device pin resources are strictly limited and the system is less sensitive to overall configuration time then serial access approaches should be considered. In serial approaches, a single configuration data bit is shifted into the device with each controlling clock pulse. Inside the device, bits may be loaded into a register for wider parallel programming but each bit must be shifted into the device, one at a time. This typically results in slower overall configuration times.

We will now examine each of the general approaches in more detail.

3.1 Parallel Access

No standard parallel access mechanism exists for PLDs. This means that vendors have developed proprietary approaches that benefit their particular devices. Parallel approaches exist for both volatile and nonvolatile devices.

The generic parallel access approach pictured in Figure 3-4 involves a two-way data bus, an address bus, some configuration control signals and sometimes, for nonvolatile devices, special voltage pin.

The data bus need only be two-way if the device allows reading and writing data. In the case where the device allows data writes only (and a control signal signals success or failure of the transaction), the data bus may be one-way.

Nonvolatile devices may need a special voltage pin to provide overvoltages to program the device. These are typically never needed for volatile devices.

The address bus may be optional in devices that maintain control over their own address function during programming. Some devices encode the address in the data stream removing the need for a separate address bus.

Nonvolatile devices typically multiplex the functionality of the parallel access pins with regular IO pins on the device. The device enters parallel mode by raising the special programming voltage pin to a high level (greater than 8V). When the voltage is lowered, the device pins return to their normal role.

For volatile devices (and we are really talking about SRAM devices), most devices have made an 8 bit wide data port available for high-speed configuration of devices. This practically entails dedicating 8 data pins and 2-3 control signals on the device specifically for configuration. While device manufacturers do allow you to multiplex the pins between configuration and mission functionality, setting up this split task environment measurably complicates system design. In addition, it significantly complicates designs that need system reconfigurability. It is also true that parallel approaches typically can only address one device at a time. In systems that are made up of many programmable devices, the routing of the configuration bus must be considered. It may contribute to a more complicated board layout.

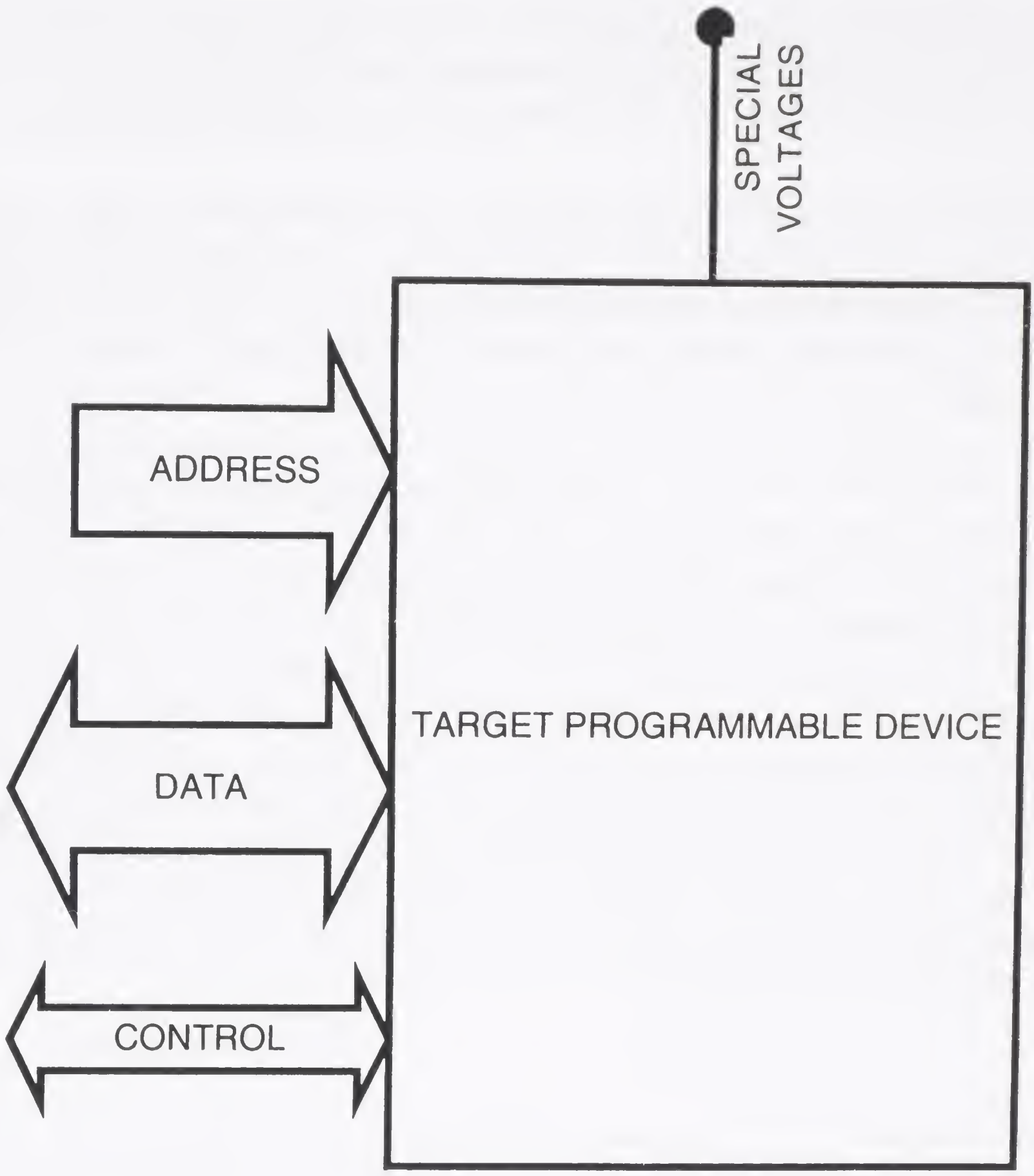

Figure 3-4. Parallel Access Diagram

For nonvolatile devices, parallel approaches are typically used when programming individual devices in socket programmers. Historically this approach was needed to facilitate application of special programming voltages to the device being configured. In addition, some voltages need to be pulsed during device configuration. Since the device is inserted in the socket programmer specifically for configuration (and not to run in mission mode), the device pins are used to apply the configuration address, data and synchronization signals. Typically, the device senses the programming voltage and goes into configuration mode in which the device pins allow direct access of the configuration control logic.

In the last ten years, this approach has fallen out of favor as devices have had more of the configuration control logic integrated into the device. This matched up with a movement towards serial access approaches and in-system configurability for nonvolatile devices. The implication for socket programmers has been simplification of their functionality. No longer need socket programmers provide unusual programming voltages, nor need they have access to all device pins. In addition, the configuration algorithms themselves have been simplified as a result and the processing power of the programmers could be reduced substantially.

3.2 Serial Access

The serial access mechanisms available are of two basic types – standard and proprietary. Often both mechanisms are available on a single device. For instance, Xilinx FPGAs support both IEEE STD 1532-based configuration and their own 4 pin serial configuration port.

The number of device pins associated with either serial approach is usually the same. Four (or sometimes five) pins are dedicated to configuration. Some manufacturers allow the configuration pins to be used as ordinary IO. As with the parallel access mode above, this approach can be dangerous. It can lock out future reconfiguration of the device. This might seem like a good idea at design time but when a customer calls with a design problem, you will have wished otherwise. Electrical noise or stray voltage spikes can accidentally trigger the mechanism to turn the IO pins back into configuration port pins.

The proprietary serial access mechanisms are popular with volatile devices that need to connect to an external nonvolatile program store. SRAM-based device manufacturers simplify connection of their devices to serial programmable read only memories (SPROM) using similar (but different) proprietary serial mechanisms. These serial connections allow for daisy chaining to both programmable devices and SPROMs to allow designers to have a central nonvolatile storage area for all devices in a system.

Figure 3-5 details a typical serial interconnect technique. Configuration data is serially shifted into the target device using an externally provided clock. The configuration data is stored in the first device until its configuration memory is full. If data continued to be clocked into the device, it is passed onto the next device in the serial chain.

The clock can be connected to all devices in the chain or passed from one device to the next, as data becomes available to pass on to the next device.

When the devices have configured successfully they signal completion using COMPLETED. COMPLETED can checked from each device individually. Alternatively, the COMPLETED signals can be connected to one another to form a wired AND connection to signal completion only when all devices have successfully configured.

Extra control signals may be provided to reset the device or to signal failure status information to allow for better diagnostics.

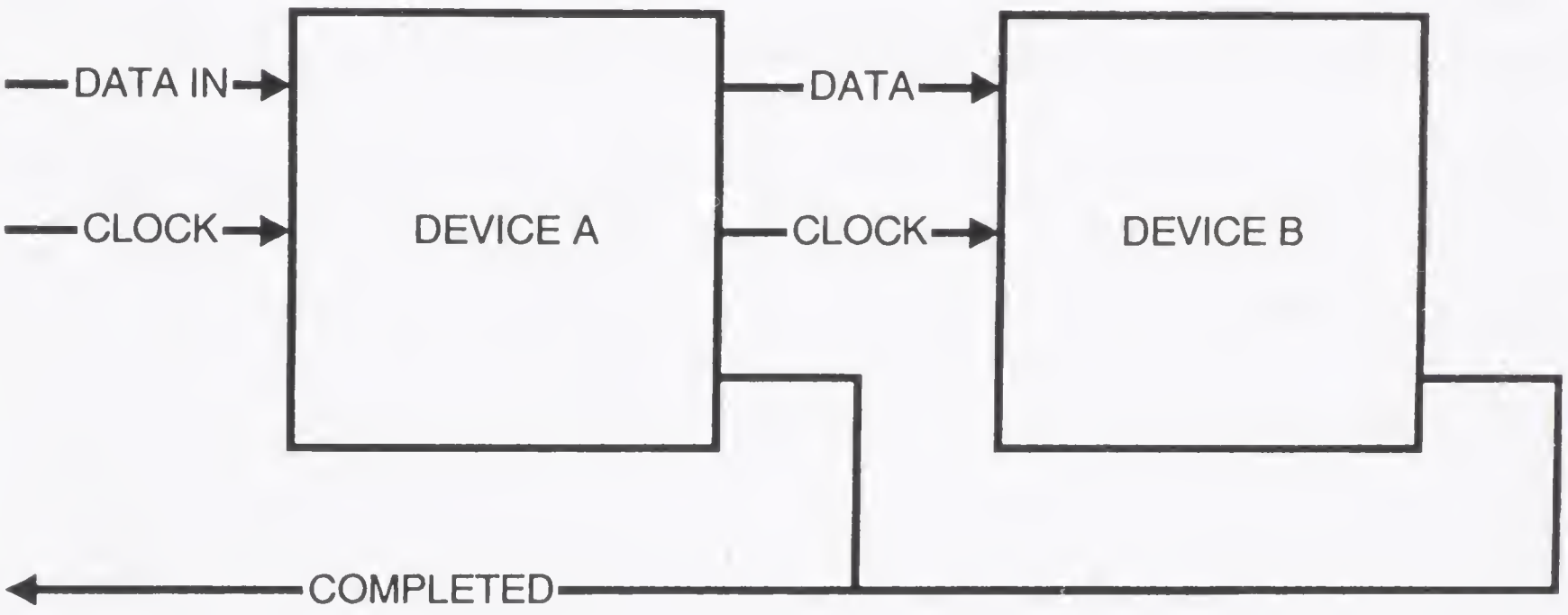

Figure 3-5. Proprietary Serial Access Diagram

If a systems designer mixes devices from different manufacturers and plans to use the proprietary serial mode then they must use separate serial daisy chains for each manufacturer. Sometimes different generations of devices from the same manufacturer may not be able to coexist in the same serial daisy chain. Designers should watch for this situation. Separate chains mean separate infrastructure to support each chain, which complicates the prototyping and manufacturing flows and increases total system costs. If they are using IEEE STD 1532-based approaches then that penalty does not apply.

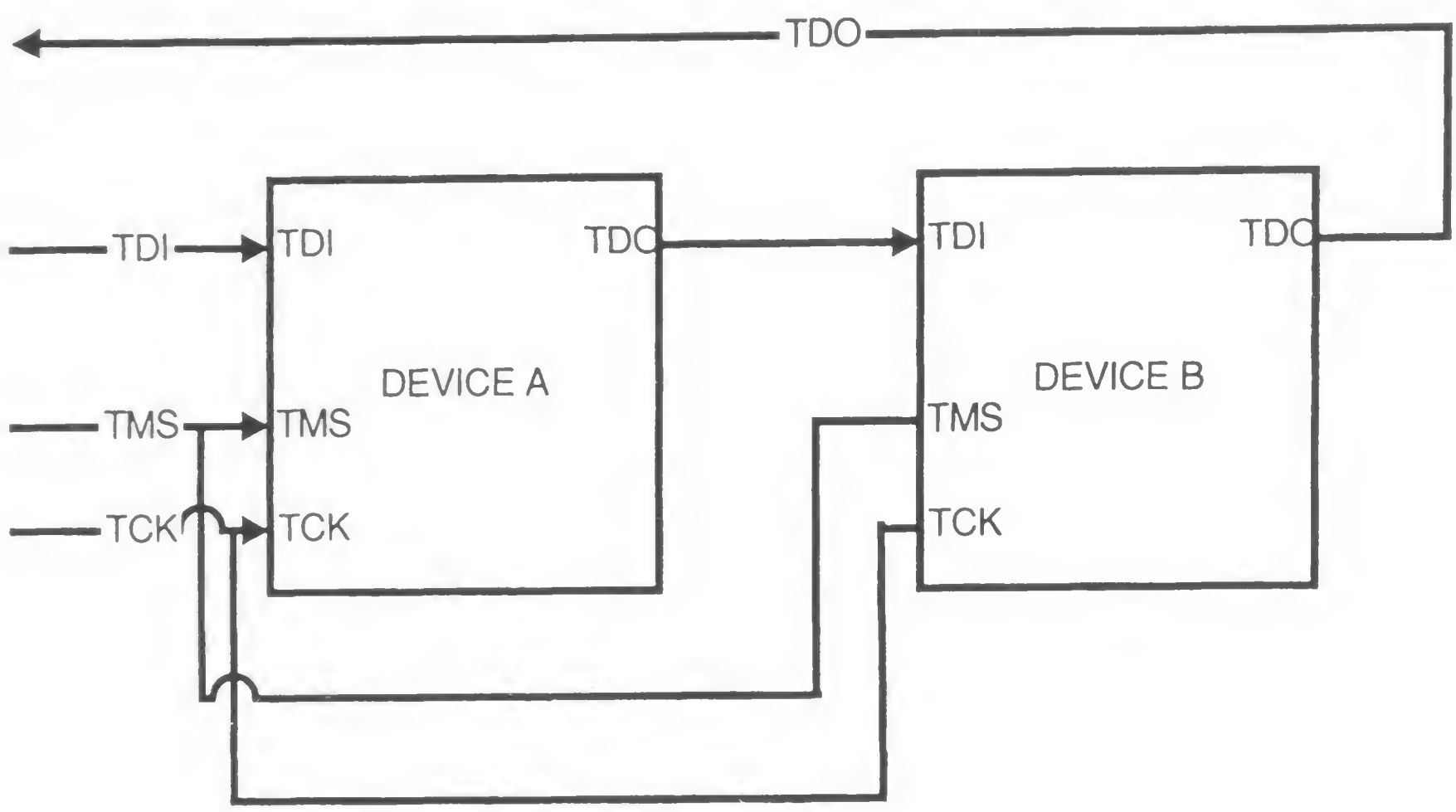

Figure 3-6. IEEE STD 1532 Serial Access Diagram

The standards-based approach is to use the IEEE STD 1149.1 test access port (TAP) of IEEE STD 1532 compliant devices to gain access to the configuration control logic. The connections are shown in Figure 3-6. They are identical to the connections required for IEEE STD 1149.1.

The TAP is usually a dedicated set of pins since it serves both configuration and test roles. Effectively there is no added pin penalty for using this access mechanism since these pins are already used for IEEE STD 1149.1 test. If a system designer either finds or develops a suitable application to drive the TAP, it is possible to configure devices from all manufacturers in a single serial chain. We will discuss approaches to provide a universal solution of this sort in later chapters.

Chapter 4

CONFIGURATION DESCRIPTION AND SPECIFICATION LANGUAGES

Configuration Data Specification

1. Introduction

Initially the increasing popularity of programmable logic devices was attributable to two developments. For CPLDs, their low cost and ease of use allowed them to be a cost-effective alternative to discrete logic. Yet, the CPLDs' need for externally applied high voltages to effect reprogramming limited their reprogrammability to prototyping applications. During prototyping were the devices affixed to the board in sockets to simplify their easy removal for reprogramming.

For FPGAs, their rapid reprogrammability but rather high cost led to their use as an excellent rapid prototyping platform. Then when production began, ASICs would replace the FPGAs to reduce the overall system cost.

The past five years have seen many technological advances in the PLD marketplace. In this period, nearly all CPLDs introduced featured in-system configurability. This allowed programming of devices at system voltages reducing the need for unusual voltages for configuration.

In addition, both the price and gate density of FPGAs improved to allow their consideration as a practical alternative to ASICs, even in production.

Another significant advance has been the widespread adoption of the communications protocol and control logic associated with IEEE STD 1149.1 as the method for controlling in-system configuration operations.

Reduced price, higher densities and a simple communications protocol have together launched reconfigurability as a valuable feature of programmable logic devices. Exploiting this feature needs an automated

manner in which to specify how to configure a device and with what data to configure them. To that end, there have been several high-level languages and data description formats proposed. So far, only two have seen widespread industry adoption. We will examine all these formats since all have some measure of popularity. We will end with a discussion of IEEE STD 1532. The IEEE STD 1532 is the approach with the most momentum and industry acceptance.

2. JEDEC Standard Data Transfer Format

This file format is an ASCII file most commonly known as the JEDEC file format (which unfortunately sells the JEDEC organization short for they have shepherded development of hundreds of standards and file formats). The JEDEC file format is formally known as *JEDEC Standard JESD3-C, Standard Data Transfer Format Between Data Preparation System and Programmable Logic Device Programmer.* It is available from the Joint Electron Device Engineering Council (JEDEC) at no charge for individual use or for a fee for redistribution.

The formal title best describes the key goal of this file format. It is chiefly a data format. It does not define any information about the algorithm to program a device. It does not define a communications protocol. It does define a data format for specifying the programmed states of the programmable device's configuration memory (in the nomenclature of JESD3-C this is the fuse information). The standard also specifies a format for defining simple functional test vectors for application to the device. The motivation for this was the concern that with increasing device complexity a mechanism for verifying the device is functioning as expected, would be invaluable.

Note the last revision to this standard occurred in 1994. The state of programmable device technology, the density of PLDs and the general acceptance of these devices were different then. Nevertheless, the JEDEC file format has endured (for CPLDs), to describe the device programming data. The use of the functional vector specification has fallen out of favor as programming devices and technology has become much more reliable. As device densities increase and the data captured in a JEDEC file increases, it is likely that this file format will fall out of favor and be replaced.

Let us now examine the basic JEDEC file.

2.1 Basic File Organization

The JEDEC file is a list of fields bracketed by a start of transmission character (STX) and an end of transmission character (ETX). Immediately following the EXT character is a 16-bit transmission checksum that is a sum of all character values between the STX and ETX. This checksum allows for some error detection. It can detect when the file has been tampered with or garbled in transit from one location to another.

All fields in a JEDEC file start with a letter. End of field data is described by a "*". The key fields in a JEDEC file are the L, C and V fields.

2.1.1 The L Field

The L field is the field that contains the fuse data information. This specifies how the program memory of the PLD should be configured to affect the functionality requested by the designer. Because the JEDEC file is based on PROM requirements, it assumes that address space of the PLD is contiguous and fully populated with programmable cells. The L field starts with an address designator, then after a space is a list of binary fuse values to be programmed, marked by a 1 or a 0. The fuse values are programmed from the address designated. The fuse field terminator is a *. If the next record specifies an address that is not the immediate next address, then all fuse locations until the mentioned address are filled with a default value (denoted elsewhere in the JEDEC file). To understand this better, consider the example in Figure 4-1.

```
L0000 01010101 10101010 11111111*
L0030 10101010 00000000 01010101*
```

Figure 4-1. Sample L Field

The first L field says the fuse addresses start at location 0 with 01010101 and the rest of the fuse values follow with 10101010 starting at location 8 and 11111111 starting at location 16. However, note the next L field address value is 30 but the previous fuse specification ended at address 23. This means, since the address space is always contiguous, that locations 24 through 29 will be filled with a default value. The default value, which can be either 0 or 1, is the same value for the whole file and is specified by the F field. If the default were 0 then the next locations would be programmed as 000000. After that, the value 10101010 is programmed into location 30.

2.1.2 The C Field

The C field provides a fuse checksum. The checksum value that follows the ETX represents the sum of all characters in the file. On the other hand, the fuse checksum represents only the value of the sum of the fuse values as represented in the L fields (incorporating any implicitly applied default values) but not including the address values. In the example above the checksum would be calculated over 01010101 10101010 11111111 000000 (the default values) 10101010 00000000 and 01010101. This checksum is often used as a quick identifier of device programming.

2.1.3 The V Field

The V field is the vector field. This specifies the set of functional test vectors applied to a device to test its functionality. The V field consists of two parts. The vector number that appears beside the V identifies the vector. This decimal number identifies failing vectors, syntax problems, or the like to the end user. Following the vector number and separated from it by a space is the vector information itself. The vector itself is a set of alphanumeric characters that represent logic values applied to or sensed on the device pins. The characters represent steady logic values applied like 0 or 1 or pulses and edges. Output values are logic one, zero or high impedance states.

2.1.4 Other Fields

As well as the previous fields, there are about 15 other fields that can specify unique programming options, device size and certain pin grouping for sophisticated testing. It is also worth noting that a comment field is available as well. Any field beginning with an N is a note and is used for descriptive or explanatory comments.

2.2 Using JEDEC Files

Recall that this file format has only programming data in it. Therefore certain assumptions made about the target device can result in a circumstance in which even though the format is standard, the interpretation is not. For instance, needing a contiguous address space based on all programmable devices being PROM-like in their memory layout is typically not true for modern devices. This means the address values named in the JEDEC file become irrelevant and the program data addresses need to be calculated outside the file.

In the end, this reduces the JEDEC file to a container for storing configuration data. The manner in which the data is interpreted and applied to the device needs customization for each device. The algorithm underlying this customization is not available in the JEDEC file itself.

Another issue with JEDEC files is the inability to store data for large devices efficiently. The file represents all device data as 1 and 0 characters. The single method for data compression allowed is specifying a default fuse value. This tells to which value to set unspecified fuse address locations. This method works acceptably well when there are long consecutive sequences of 1's or zeroes in the device address space. If the sequences of 1's or zeroes are moderate in length then more L records are needed which might increase the file size. In addition, a contiguous address space is necessary for this to work correctly.

In summary, the JEDEC file, that served us so well for so long, is nearing the end of its useful life. In addition, because it is a data only file format, it does not serve the reconfigurability application space well since the algorithm needs to be custom developed for each device.

Chapter 5

CONFIGURATION DESCRIPTION AND SPECIFICATION LANGUAGES

Configuration Algorithm with Data Specifications

1. Serial Vector Format

Serial Vector Format (SVF) is an ASCII file format designed to promote the exchange of boundary-scan data between development systems and boundary-scan hardware. It was developed jointly by Texas Instruments and Teradyne. Control over the file format has since been handed off to boundary-scan solution provider ASSET InterTech. The most recent revision of this file format is Revision E from March 1999. Copies of the SVF specification are available at no charge direct from ASSET InterTech.

The format can be thought of as assembly code for boundary-scan in that it defines the low-level simple state machine operations typically associated with running device and interconnect tests. It also mixes the test algorithm and the test data. Of late, the format has also been used to describe device configuration. While there are many subtleties of the format, it is a file that contains boundary-scan commands and data. The key commands are SIR, SDR and RUNTEST. We will describe the essentials of SVF format using the example below.

1.1 SVF File Structure

In the coming sections, we will examine the commands that make up an SVF file.

1.1.1 The SIR Command

In using a boundary-scan device, the 16-state boundary-scan state machine is traversed. The roles of each state are well defined. For instance, the Shift IR state shifts instruction data into the device's instruction register.

After instruction loading, associated data (if any) can be loaded and device operation execution completed. The purpose of the SIR command is to direct the state machine to transition to the Shift IR state and load the specified instruction data into the instruction register.

```
ENDIR IDLE
SIR 8 TDI(EA)
```

Figure 5-1. SIR Command Example

In Figure 5-1, the SIR command directs the data EA (represented in hexadecimal) into the device instruction register. The rightmost bit shifts into TDI first. The ENDIR command says that after the shifting is complete the Run Test/Idle state should entered. This is true for all SIR commands that follow the ENDIR command.

The SIR command can also optionally test data sensed on TDO as data shifts in to TDI. In addition, input and output data masking can point out which bits are significant.

1.1.2 The SDR Command

Similar to the SIR command, above, the SDR signals the specified data is shifted into the data register when in the Shift DR TAP controller state. This command typically follows an SIR that sets up the target data register. The data associated with the active instruction is then shifted in using the SDR command.

```
ENDDR IDLE
SDR 16 TDI(A5C5)
SDR 16 TDI(0000) SMASK(0000) TDO(00E0) MASK(FFFF)
```

Figure 5-2. SDR Command Example

In Figure 5-2, there are two SDR commands. The first SDR command shifts in 16 bits of data associated with the previous SIR command. The data (as before) is represented in hexadecimal. Also, as before, the rightmost bit of the input data (A5C5) is shifted in first. Similar to the ENDIR command, there is also an ENDDR command that points out to which state the state machine should transition after completing a data shift in the Shift DR tap controller state. In our example, the command says that Run Test/Idle is the end state.

The second SDR command shows how it can test data shifted out the device on TDO. Once again there is a 16-bit shift (since this is the size of the target register). In this case, however, the data shifted in on TDI is don't care data. Although the TDI data is all zeroes, the SMASK signals (since it too is all zeroes) that none of those bits are significant (a 1 points out significant bits in the associated bit position of the SMASK). This give the system that is applying the stimulus the choice of producing arbitrary values on input since they are all don't cares.

The understanding is different however on the expected value side. The data expected on TDO is 00E0 (where as before, the rightmost bit represents the data value first seen shifted out on TDO). The MASK field says that all values seen on TDO are significant. Once again, a 1 marks significant bits in the associated position of the MASK. Since the MASK value is all 1's (FFFF), all values are significant and all TDO values shifted out must match exactly with those named in the TDO field.

1.1.3 The RUNTEST Command

Many boundary-scan actions and most every configuration operation need that some time to pass for the procedure to complete. The IEEE STD 1149.1 TAP state machine has a state typically used for this expressed purpose. Test (like built-in self-tests) execution, signal settling (say, when using INTEST) or device programming and erasure typically completes in the Run Test/Idle TAP controller state.

The RUNTEST command allows description of just this operation. It can signal either an absolute time spent in Run Test/Idle or a specific number of TCK pulses. In addition, the command has the added flexibility of being able to specify a TAP controller state to which to transition after completion of the appointed wait period.

SDR 16 TDI(A5C5)
RUNTEST IDLE 32 TCK

Figure 5-3. RUNTEST Command Example

In Figure 5-3, the RUNTEST that follows the first SDR command says that 32 TCK pulses should occur in Run Test/Idle. Since no exact end state is named, after completing the mentioned pulses the state does not change.

1.1.4 Other Commands

There are wide varieties of other commands that are available. These include:

- Commands for handling parallel IO pins (rather than just the boundary-scan TAP pins)
- A command to effect arbitrary TAP state machine transitions
- A command to control the optional TRST pin of the boundary-scan TAP
- A command to set the TCK frequency

Because boundary-scan devices typically connect in serial chains, a set of commands is available to specify fixed data prefixes and suffixes shifted in before and after data mentioned in the file's SIR and SDR commands. This allows quick customization of an SVF file produced targeting a single device by itself to target the device in a fixed serial chain.

1.2 Using SVF Files

SVF files' original use was description of basic boundary-scan test sequences for test hardware like automatic test equipment and PC-based tools. The point was to develop a format easily produced by test software for use on a multiplicity of test platforms. It can be thought of as assembly code for boundary-scan tests.

The generation of SVF files for configuration is typically from vendor-supplied tools that examine the configuration data contained in, say, JEDEC files. Then, by having specialized knowledge of the configuration algorithm, the vendor tool transforms the configuration data into the SVF statements that properly sequence the information into the device.

When you want to run an SVF file, you can do it in one of two ways. The file can be directly interpreted and executed by a software or hardware interpreter. This is simplest approach but has the drawback of tending toward slower execution times and needing memory proportional to the size of the file. Straight interpreted solutions are simple to implement and are often available as shareware.

Another approach is to first compile the SVF into format more suitable for direct execution on the target hardware and potentially including memory and run time optimizations drawn from examining the target SVF file. This

results in faster and more efficient execution of SVF files. Because of their relative complexity, you must buy solutions of this sort typically from boundary-scan tools developers.

There were several assumptions made that were suitable for testing but not so for configuration. Among the assumptions made were the following:

- The target tests are short sequences of (potentially) long vectors.
- The target tests are go/no go tests with no need for complex program control flow.
- The target tests have no run time dependencies.

These assumptions are reasonable for interconnect and device tests but when applied to device programming they fall a bit short. In particular, vector sequences that configure devices can have long sequences of potentially long vectors. This makes the SVF file large. In fact, the file is much larger than expected by the original design of the SVF specification.

In addition some legacy configurable devices (for example, those based on flash technology) have the characteristic that erase and often program operations are non-deterministic. This means that even though instruction and data sequence correctly to erase or program the device, it may be the case that the process may need a variable number of retries of the same operation to complete successfully. SVF does not have this feature as part of the language.

Some legacy configurable devices have configuration algorithm parameters (like erase times, program times and sometimes the algorithm flow) depend on data read out of the target device. SVF does not have a mechanism for reading data out of the device and using it as part of the SVF file.

Despite these limits, SVF because of its broad acceptance in the test community has been able to serve as a useful method for describing most device configuration algorithms. The matter of file size remains an issue with SVF since data is represented in ASCII hexadecimal.

To address the limits, some vendors who produce SVF to describe device configuration have added proprietary commands in comments in the SVF file which when correctly interpreted result in faster and more efficient device configuration. Others have proprietary rules for interpreting the produced SVF for devices to configure more efficiently. In either case, SVF

executed without the special commands or according to standard interpretation the devices will still configure.

2. STAPL - Standard Test and Programming Language

To address the limits of SVF and provide a platform that better solved the issues associated with configuring programmable devices on embedded processors, Altera Corporation put forward a proposal high-level language known as JAM. Not an acronym, only a name, JAM incorporated all the functionality and the essential syntax of SVF and wrapped a BASIC-like program control flow around it. In addition, it standardized a data compression format for use in the language. JAM was submitted to JEDEC for standardization shortly after its first proposal.

With the support of JEDEC committee JC42.1, over the course of several years, JAM was changed, improved and standardized. The new standard version of this language became known officially as JEDEC Standard JESD71 Standard Test and Programming Language (STAPL). As any JEDEC standard, the STAPL specification is available free for personal use from JEDEC or for a fee for redistribution.

Support for STAPL is spotty with some semiconductor vendors supporting wholeheartedly (for example, Altera), others tepidly (for example, Xilinx) and still others not at all (for example, Lattice Semiconductor). A similar situation exists in the tool vendor space.

Since the STAPL is to some extent based on SVF, the focus of the description will be on the differences.

2.1 Basic STAPL File Structure

Because the STAPL file describes multiple distinct and distinguishable functions (for example, erase, program, verify) in a single file, it is a more sophisticated language than SVF or a regular JEDEC file. Because STAPL includes both data and algorithm (but with some degree of separation), it has a more formal structure. To understand STAPL files it is important to understand the overall file structure first before going into the details.

A STAPL file is comprised of a sequence of NOTE statements, ACTION statements, PROCEDURE and DATA statements and then finally a CRC statement. The order of the statements is exactly as shown.

NOTE statements contain text strings that can identify the contents and features of the STAPL file. NOTE statements are not executable – they are purely informational.

A STAPL file must contain at least one ACTION statement. The ACTION statements match the operations that are available to end users of the target device. Examples of ACTIONS would be erase, program or verify

Each ACTION is in turn comprised of one or more PROCEDUREs. The PROCEDUREs associated with an ACTION are listed in order of execution. An example of this would be that if an end user specified the ACTION "program", it might consist of the PROCEDURE sequence of erase, followed by program, followed by verify. A further flexibility is that PROCEDUREs can be optional or recommended allowing the user to enable or disable their execution in a specific session of the STAPL file. Each PROCEDURE contains executable statements that include all the basic functionality of SVF (like loading data into and out of the instruction and data registers). Instead of using SIR and SDR, STAPL uses IRSCAN and DRSCAN. The syntax of the commands is slightly different but immediately obvious to those familiar with SVF. STAPL however extends SVF to include control flow statements like "IF <condition> THEN" and "FOR" loops. In addition, STAPL allows variables to read and store data from the device. This data can be tested against an expected value. Such testing can direct program execution along different paths.

The DATA block contains variable declaration statements. PROCEDURES can only use variables in DATA blocks if the PROCEDURE signals this through the USES keyword. DATA blocks separate DATA that is likely to be updated from other program data.

The CRC statement contains the cyclic redundancy code of the entire STAPL file. It verifies the overall integrity of the file.

In keeping with the intended strict application space of device configuration (and test), the standard did not define any improved features that might turn STAPL into a more general purpose programming language or unnecessarily burden implementation.

An example of this is that each STAPL file is a standalone application. The standard does not allow for linking multiple STAPL files together to share assigned variable space or procedure calls. So a STAPL file that describes the configuration algorithm for Device A and the STAPL file that described the configuration algorithm for Device B will be performed sequentially. The presumption is that each application will have its resources freed up after execution. In addition, STAPL has no string variables (although it has string constants), no floating-point variables or arithmetic.

A feature key for programmable logic was including a data type to represent and store compressed data. The compression technique referred to as the Advanced Compression Algorithm (ACA), looks for repeated sequences of groups of 8 bit data patterns in a data stream. Identified sequences that repeat are represented as compressed data by referring to their first encountered location offset in the data stream rather than the actual data. This compression technique works well when data is not too random (for example, not encrypted) and when the data stream does not contain address information (which because it increments breaks the pattern algorithm of ACA).

STAPL has limited user input and output functionality. It allows message printing using the PRINT statement and the display of integer values to the end user using the EXPORT statement.

These limits make sense given the scope of the problem the language intended to solve.

2.2 STAPL File Example

To complete understanding of the operation and use of a STAPL file, it is useful to work with a simple example. The sample file in Figure 5-4 helps explain the basic functionality of STAPL.

```
'Set up NOTE fields with generation,
'run time and device information
'None of this information is executable
'None of this information is used by the STAPL program
NOTE "CREATOR" "STAPL Generator 5.2.3"
NOTE "DEVICE" "TEST32MC";
NOTE "DATE" "1997/12/31";
NOTE "STAPL_VERSION" "JEDS00-A";
NOTE "ALG_VERSION" "3";
NOTE "STACK_DEPTH" "2";
NOTE "MAX_FREQ" "10000000"; '10MHz
NOTE "TARGET" "1";
NOTE "IDCODE" "00FDEC01";
'Beginning of executable portion of file
'Define ACTION for file
ACTION READ_IDCODE = DO_READ_IDCODE;
'Define PROCEDURE used
PROCEDURE DO_READ_IDCODE;
'Declare variables for data arrays
BOOLEAN capture_data[32];
BOOLEAN idcode_instr[9] = #001101000;
BOOLEAN all_ones[32] = $FFFFFFFF;
INTEGER i;
'Initialize device by going to Test Logic Reset
STATE RESET;
'Load idcode instruction
IRSCAN 9, idcode_instr[8..0];
'Capture idcode shifted out of device
DRSCAN 32, all_ones[31..0], CAPTURE capture_data[31..0];
'Display captured value on console
EXPORT "IDCODE", capture_data[31..0];
ENDPROC;
'File CRC
CRC 3759;
```

Figure 5-4. Sample STAPL File

The task described by the file is contrived. The ACTION named "READ_IDCODE" when invoked calls the PROCEDURE named "DO_READ_IDCODE". The PROCEDURE named "DO_READ_IDCODE" sends the TAP state machine to the Test Logic Reset state and then directs loading device's IDCODE instruction. After loading the IDCODE instruction, the IDCODE value itself may be shifted out of the device. In this PROCEDURE, the value is shifted out 32 times. After each shift the first bit out is tested to see if its value is logic '1'. IEEE STD 1149.1 requires the first bit of the IDCODE value be a logic '1'. If it is not a logic '1' there are two possible reasons. The first is the device is designed incorrectly and the IDCODE value does not adhere to the standard.

While this was a real possibility a decade ago it is unlikely now that IEEE STD 1149.1 is well publicized and understood. The second, and more likely, possibility is there is a signal integrity problem or a bad connection between the device and the system executing the STAPL application. This loop therefore provides a crude system integrity test. If the test passes 32 times in a row, the execution completes with a success status and the IDCODE value is returned to the console application. If the test fails once, the "IDCODE read incorrectly" error status is returned and the wrong IDCODE value as read is returned to the console application.

The NOTE information contained at the top of the STAPL file contains data suitable for display to the end-user as well as data that are valuable for the interpreter itself to promote more optimal processing of the file. The NOTE data includes information like:

- The maximum depth of the call stack
- The maximum clock frequency of TCK
- The number of devices included in the boundary-scan chain accessed by the STAPL file.
- The IDCODE of the device in the STAPL file.

These values can be useful to the interpreter in setting up the run-time environment, pre-allocating memory and deciding if the expected operating conditions can be met on the execution platform. While there is no requirement in the standard that the interpreter validate these values, it is valuable to use one that does.

2.3 Using STAPL Files

By building on SVF, STAPL provides similar functionality but includes features suitable for programmable logic devices. This extra functionality included control flow and data compression. The target platforms included those of SVF (automatic test equipment and PC-based tools) but also STAPL intended to address the embedded processor space. As with SVF, a key need was developing a format easily produced by test software that was usable on a multiplicity of test platforms.

There was a vision that STAPL files would supplant the existing JEDEC format as the device program file output from PLD design tools. While some vendors took steps to begin the effort to realize that goal, others were less enamored of the solution. They were concerned about several key issues:

- The general effectiveness of the data compression algorithm
- The difficulty of updating configuration data or the configuration algorithm separately
- The applicability of the solution to resource limited embedded systems.
- The large run-time memory appetite of the approach

Experimental results pointed out that ACA format would not afford high compression for their data formats (for reasons previously told)

The produced STAPL files would have to include algorithm and programming data. In the case in which the algorithm or program data changed separately, new end-user procedures would need to be in place that were burdensome and complicated for the end-user.

In targeting the embedded system environment, the basic STAPL interpreter was large. It would not fit in the code space of available 8 bit micro controllers (these were key platforms that needed support then). In addition, the STAPL interpreter was significantly slower than the available customized solutions.

The interpreter's run time memory needs were excessive in simple-minded implementations since the compressed data needed to be fully decompressed at run time. This means the run time memory needed equals the compressed data size plus the uncompressed data size. This effectively incurs a memory penalty for using compression. To provide a more intelligent approach for end-users would be obliged a support burden in added system memory costs.

To address these issues, after standardization some vendors (notably Altera) spent some effort developing a customized embedded solution. These solutions supplemented the STAPL standard by compiling a STAPL file into a proprietary byte code format. This byte code format could then run on a smaller and more efficient interpreter. The results were much better in both run time and memory consumption. The end-user that wished to use the byte code format would get the "compiler" to create the byte code file from the STAPL source file. Then they would get the byte code interpreter and customize it to their intended platform. Neither the compiler nor the byte code itself was published as part of the STAPL standard so they remained proprietary technology. This acted as a barrier to wider acceptance of the byte code approach.

The typical flow now used to produce a STAPL file is similar to that of the SVF file in the best case and more complicated in the worst case. In the usual (and best) circumstances, vendors provide applications that read configuration data stored in, say, a JEDEC file. Then by having specialized knowledge of the target device's configuration algorithm, they are able to produce a STAPL file containing both the specific sequence of configuration instructions needed and the data in the ACA compressed format.

In less ideal conditions or in conditions in which the vendor has no direct STAPL support, SVF files are produced as previously described and these SVF files are then translated to STAPL format. This path often results in less than optimal STAPL files that are larger and slower than they would be, had STAPL been directly output.

One less desirable but still possible generation scenario is to write the configuration algorithm by hand in STAPL and then attach the data in ACA format to the handwritten STAPL algorithm.

If STAPL byte code is produced then a further translation step is always needed. This output is only as efficient as the STAPL source input to it.

In the same way that SVF was optimized for testing, one might argue that STAPL was optimized for programming.

The test specific functionality in the STAPL standard is optional. Specifically, the functions used independently to set up non boundary-scan pin values are not part of the mandatory standard implementation.

The "configuration only" functionality merely adds overhead to test implementations.

Of course, if you intend to integrate device configuration and test then these issues do not apply.

Chapter 6

CONFIGURATION DESCRIPTION AND SPECIFICATION LANGUAGES

Separated Configuration Algorithm and Data Specifications

1. Java API for Boundary-Scan

While STAPL was in development, there were concerns raised about STAPL's flexibility, infrastructure support and the ability of STAPL to develop into a practicable cross platform solution. To address these issues developers proposed a new Application Programming Interface (API). An API is library of related routines that provide well-defined and fully specified access to a particular product or feature. In this case, the feature is the device's boundary-scan test access port. The idea was to leverage the infrastructure of an already proven technology that featured true portability, broad-based tools support, widely available platform support, a broad knowledge base and true scalability. The single programming language that delivered all those characteristics was Java.

In addition, basing development on Java meant the immediate availability of a vast reservoir of Java libraries that could ease such necessary functionality as remote connectivity (for system update), security (for transmission and transaction security). Because Java is an object-oriented language, standard object interfaces could be defined. Vendors, users and developers could then customize these to provide proprietary functionality in a standard way. There is a suite of certification tests that all Java platforms must pass. These are managed by Sun Microsystems to ensure that all Java platforms will behave identically. The problem of platform certification is therefore independent of the API.

1.1 Java

Java is an object-oriented programming language developed by Sun Microsystems. It shares many superficial likenesses with C and C++ (for

instance, for loops have the same syntax in all three languages). It is not based on any of those languages, nor have efforts been made to make it compatible with them.

Java was originally created because C++ proved inadequate for certain tasks. Since the designers were not burdened with compatibility with existing languages, they were able to learn from the experience and mistakes of previous object-oriented languages. They added a few features C++ doesn't have like garbage collection and multithreading; and they threw away C++ features that had proven to be better in theory than in practice like multiple inheritance and operator overloading.

Even more importantly, Java's ground-up design allowed for secure execution of code across a network, even when the source of that code was unknown and possibly malicious. This required removing more features of C and C++. Most notably, there are no pointers in Java. Java programs cannot (at least in theory) access arbitrary addresses in memory.

Further, Java was to be cross-platform in source form, but also in compiled binary form. Since this is impossible across processor architectures, Java is compiled to an intermediate byte-code that is interpreted at run time by the Java interpreter. Thus porting Java programs to a new platform only needs a certified Java interpreter on the target platform.

In addition, Java has several features to make programming bugs less common:

- Strong Typing
- There are no unsafe constructs
- The language is small so it's easy to become fluent
- There are no undefined or architecture dependent constructs
- Java is object oriented so reuse is well-supported

1.2 Where did Java come from?

In the late 1970's, Sun Microsystems' founder Bill Joy thought about doing a language that would merge the best features of MESA and C. However, other projects intervened. He picked up the idea again in late 1990 and wrote a paper that outlined his pitch to Sun engineers that they should produce an object environment based on C++.

Around this time, James Gosling (who developed "emacs") had been working for several months on an SGML editor called "Imagination" using C++. Frustrated by the difficulties of using C++, he developed Oak as the implementation language for "Imagination". Oak later became Java.

Patrick Naughton started the Green Project in late 1990. The project was defined as an effort to "do fewer things better". He recruited Gosling and Mike Sheridan to help start the project. Joy showed them his paper, and work began on graphics and user interface issues for several months in C.

In April of 1991, the Green Project (Naughton, Gosling and Sheridan) settled on smart consumer electronics as the delivery platform, and Gosling started working in earnest on Oak. Gosling wrote the original compiler in C. Naughton, Gosling and Sheridan wrote the runtime-interpreter, also in C. Oak was running its first programs in August of 1991. The first demos of that system were given in the winter of 1991.

By the fall of 1992 "*7", a cross between a PDA and a remote control, was ready. Following a successful demonstration, the Green Project was set up as First Person Inc., a wholly owned Sun subsidiary.

In early 1993, the Green team heard about a Time-Warner request for proposal for a set-top box operating system. First Person quickly shifted focus from smart consumer electronics to the set-top box OS market, and placed a bid with Time-Warner.

They lost the bid and in the end the Time-Warner project went nowhere. First Person continued work on set-top boxes until early 1994, when it concluded that like smart consumer electronics set-top boxes were more hype than reality.

Without a market to be seen, First Person was rolled back into Sun in 1994. However, around this time it was realized that the requirements for smart consumer electronics and set-top box software (small, platform independent secure reliable code) were the same requirements for the nascent web.

For a third time, the project was redirected, this time at the web. A prototype browser called WebRunner was written. After more work, this browser became HotJava.

1.3 Java and the World Wide Web

Java is, chiefly, a programming language. The original use of the Java language (set-top boxes) needed security and the ability to execute code from untrusted hosts. It turns out these are the same requirements for allowing people to download and run programs from the Web. No other language has the built-in security of Java. In addition, because web programs can be downloaded on a multiplicity of platforms, cross-platform portability is also paramount. The object-oriented nature of Java is secondary, and mainly reflects the preferences and prejudices of the developers who set out to write a secure language. The C-like syntax of the language is even less crucial.

1.4 Java and In-System Configuration

Why use Java, a secure general purpose programming language with web features, for configuration of PLDs?

The challenges associated with device configuration are near identical with those associated with the specific strengths of Java.

Device configuration needs to be supported on a multiplicity of disparate platforms. These platforms range from embedded systems to PCs and workstations to Automatic Test Equipment. Great expense and effort currently is squandered in porting configuration applications from platform to platform. This is an error prone process and often results in a multiplicity of files that need to be coordinated each time a system update or revision occurs.

Device configuration is increasingly performed on devices contained in-systems that are network-connected. The network connectivity is exploited to ease field upgrade of the systems. Specialized custom software often needs to be developed to make the configuration application network accessible. This too may need to be reproduced across platforms.

Device configuration is being designed into systems as an essential portion of the systems functionality. The implication being, that during system operation, devices are reconfigured to adapt to new data input. Also, devices may be reconfigured to perform changed operations on data input to, or as data is output from, the system. This means that being able to integrate configuration software with that of the system – regardless of the platform or specific system implementation - is a necessity.

As configurable systems become networked, both the configuration software and the configuration data must be secured. The configuration software must do no harm to the system while running. The configuration data must be able to be securely transferred from source to destination.

From a practical sense, it was determined that all this would be best provided through existing technology that is well supported rather than one that is developed from scratch. For these reasons, Java appeared perfect.

1.5 Development of Java API for Boundary-Scan

Learning from the lessons of SVF and then STAPL, the Java API for Boundary-Scan (JAPIBS) was specified and built by a team of engineers at Xilinx with input from boundary-scan tool manufacturers, ATE vendors and a few other semiconductor manufacturers. The idea was to leverage the power of Java to afford greater flexibility to both producers and consumers.

The Java language has been classified into 5 separate platforms of broadening scope. They are as follows:

- Java Card
- K Java
- Embedded Java
- Personal Java
- Enterprise Java

Java Card is the version with the smallest footprint (about 32Kbytes). In addition, Java Card is the only one of the platforms that constrains the language to a specific subset (no floats, no integers).

K Java is smallest of the fully featured Java platforms. It supports all language constructs but limits the system libraries available for use. The footprint for K Java is approximately 200Kbytes and it is targets for handheld devices like PDAs and communicators.

Embedded Java is the general Java solution for the embedded space. It too supports all language constructs but allows designers to choose only the libraries necessary for their application to function. This promotes development on the smallest possible footprint virtual machine but with the exact functionality needed.

Personal Java is the Java solution that targets the Internet appliance market. This includes applications like Internet phones and remote data entry terminals.

Enterprise Java is the version of Java that you can download free for use on your PC or workstation. This includes all advertised functionality of all the available java libraries and extensions.

To guarantee broad platform coverage, the language subset of Java used for JAPIBS was constrained to that of Java Card. This meant that JAPIBS would be able to work on the largest through to the smallest of all possible platforms. This ranged from 8 bit microprocessor systems based on 8051s and other similar machines, all the way through to standard PC and UNIX workstation.

By making use of the object oriented powers of Java it is possible to allow producers of JAPIBS applications the ability to provide customized compression techniques that could be seamlessly integrated. This simply involves supplying an object declared as an interface to define the data access methods. A default implementation is provided, that can be overridden by any particular application.

The data is accessed independent of the application through the data access object and its associated methods. This allows the data to be separated from the configuration algorithm. It also allows the data to be stored in any arbitrary local or remote location and accessed as needed. By using a fully functional programming language, specific configuration applications can be easily customized by either the producer or the end user. These customizations can include such functions as polling for configuration data changes or having the configuration data changes themselves trigger configuration updates.

Java applications built using JAPIBS are commonly referred to as "scanlets". This is a play on the Java term "applet" suggesting that a JAPIBS application incorporates boundary-scan (or scan) operations.

1.6 Basic Java API for Boundary-Scan File Structure

An application developed based on the JAPIBS is built like any Java application or applet. The key difference is, of course, that when boundary-scan operations are needed, specific API calls are made. In this section, we will examine the basic set of API calls. What you will immediately notice is

the set of boundary-scan operations is universal so there will be some likenesses between these API calls and the operations included in SVF and STAPL.

1.6.1 The API Components

Given the operation of the IEEE STD 1149.1 state machine the number of operations supported by the API are simple and can be simply enumerated. We present them with objects and associated methods to follow the Java object oriented model.

Four basic classes and interfaces make up the Java API for Boundary-Scan. These four classes are:

javaScanOperations – This class describes all basic boundary-scan operations.
javaScanState – This class describes the 16 states of the TAP controller state machine.
javaScanBitIf – This interface describes the method for accessing the boundary-scan test or programming data.
javaScanHWIf – This interface describes the method for producing the electrical signals to stimulate the TAP.

1.6.1.1 The javaScanState Class

The javaScanState class describes the 16 states of the IEEE STD 1149.1 Test Access Port Controller. It is used to mirror the state of the hardware state machine. It includes defining constants for each of the 16 TAP controller states as follows:

CAPTURE_DR - The CAPTURE_DR state
CAPTURE_IR - The CAPTURE_IR state
EXIT1_DR - The EXIT1_DR state
EXIT1_IR - The EXIT1_IR state
EXIT2_DR - The EXIT2_DR state
EXIT2_IR - The EXIT2_IR state
PAUSE_DR - The PAUSE_DR state
PAUSE_IR - The PAUSE_IR state
RUN_TEST_IDLE - The RUN_TEST_IDLE state
SELECT_DR_SCAN - The SELECT_DR_SCAN state
SELECT_IR_SCAN - The SELECT_IR_SCAN state
SHIFT_DR - The SHIFT_DR state

SHIFT_IR - The SHIFT_IR state
TEST_LOGIC_RESET - The TEST_LOGIC_RESET state
UPDATE_DR - The UPDATE_DR state
UPDATE_IR - The UPDATE_IR state

In addition, a set of methods is available to set and retrieve the current state.

getState() - The getState method returns the current TAP state.
setState(byte aState) - The setState method sets the current TAP state to the state named by aState.
setState(javaScanState aState) – The setState method sets the current TAP state to the state named by aState.

1.6.1.2 The javaScanBitIf Interface Class

The javaScanBitIf interface class describes the interface methods for accessing application specific data. It is the task of the scanlet developer to provide an implementation of this interface. Completing this interface's methods must incorporate the needed data compression and decompression algorithms.

The interface specification includes definition of standard bit positions and data access methods.

BIT_0 - The BIT_0 constant provides a mask to identify the data at bit position 0
BIT_1 - The BIT_1 constant provides a mask to identify the data at bit position 1
BIT_2 - The BIT_2 constant provides a mask to identify the data at bit position 2
BIT_3 - The BIT_3 constant provides a mask to identify the data at bit position 3
BIT_4 - The BIT_4 constant provides a mask to identify the data at bit position 4
BIT_5 - The BIT_5 constant provides a mask to identify the data at bit position 5
BIT_6 - The BIT_6 constant provides a mask to identify the data at bit position 6
BIT_7 - The BIT_7 constant provides a mask to identify the data at bit position 7
EQUALS - The EQUALS constant is returned by the equals() method when equivalence is true.

NOT_EQUALS - The NOT_EQUALS constant is returned by the equals() method when equivalence is false.

The set of methods includes several overloaded method calls. These methods perform the same function but accept different data parameters.

clear() - The clear() method clears the underlying object data to all zeroes.

copy(javaScanBitIf) - The copy method copies all the bits from the javaScanBitIf value specified into the underlying application specific data structure.

equals(byte) - This equals() method tests the equivalence of the contents of the byte abyte with the underlying object.

getBit(int) - The getBit() method returns the logic value of bit stored at position i.

getBitCount() - The getBitCount method returns the total number of data bits represented in the underlying object.

getBits(int, int, byte[]) - This getBits method copies length bits to the array of byte values specified in the theBits structure from the underlying application specific data structure.

getByte(int) - The getByte method returns the byte of data stored at byte position i

getByteCount() - The getByteCount method returns the total number of data bits represented in the underlying object as a byte count.

getInt(int, int) - The getInt method returns (up to 32) length bits of contiguous data from the underlying application specific data structure as an int value.

setBit(int, byte) - The setBit() method sets the bit value specified by b at bit position i.

setBitCount(int) – The setBitCount method sets the total number of bits stored in the underlying object.

setBits(byte[]) - This setBits method copies an array of byte values specified in the theBits structure to the underlying application specific data structure.

1.6.1.3 The javaScanHWIf Interface Class

The javaScanHWIf describes the interface to boundary-scan hardware. The scanlet wiggles the TAP pins to effect configuration of the device though this interface. Because the interface will be hardware and platform dependent, the implementation is up to either the scanlet developer or the provider of the device communications hardware.

close(byte) - The close() method is used to end communications with the boundary-scan hardware interface.
getTDO() - The getTDO() method returns the current sampled value of the TDO pin.
open(byte) - The open() method is used to launch communications with the boundary-scan hardware interface.
operateTAP(int, byte[], byte[], byte[], byte[]) - The operateTAP method is used to stream an arbitrary sequence of bits to the boundary-scan hardware interface.
pulseTCK(int) - The pulseTCK() method is used to produce an arbitrary number of zero-to-one transitions on the boundary-scan TCK pin.
setTCK(byte) - The setTCK() method is used to drive the TCK boundary-scan pin to any state.
setTCKFrequency(int) - The setTCKFrequency() method sets the TCK operating frequency (if possible) to the specified frequency in hertz.
setTDI(byte) - The setTDI() method is used to drive the TDI boundary-scan pin to an arbitrary state.
setTMS(byte) - The setTMS() method is used to drive the TMS boundary-scan pin to an arbitrary state.
setTRST(byte) - The setTRST() method is used to drive the optional TRST boundary-scan pin.
waitState(int) - The waitState() method signals how long (in microseconds) to pause.

1.6.1.4 The javaScanOperations Class

The javaScanOperations class defines all the basic boundary-scan operations used by a device in defining either test or configuration algorithms. The functionality included is sufficient to allow a complete description of state trajectories and transitions for either IEEE STD 1149.1 boundary-scan test or IEEE STD 1532 configuration.

destroy(byte) - The destroy method is called to terminate and clean up resources allocated and used by the javaScanOperations object.
drEnd(byte) - The drEnd method specifies the state to which the TAP controller state machine should transition following execution of any drScan that completes in the EXIT1-DR state.

drPostpend(javaScanBitIf) - The drPostpend method specifies those bits that should be shifted in on TDI after those bits named in drScan method calls.

drPrepend(javaScanBitIf) - The drPrepend method specifies those bits that should be shifted in on TDI prior to those bits named in drScan method calls.

drScan(javaScanBitIf, javaScanBitIf, byte, byte) – The drScan method uses overloading to define various manners in which to drive the TAP controller state machine to the SHIFT-DR state. The variety of different operations includes simply shifting data in, indicating expected data to be shifted out, allowing shifting of blocks of data, skipping transitions through Run Test Idle on completion. All of these different operations are usable alone or in combination through the power of overloading.

irEnd(byte) - The irEnd method specifies the state to which the TAP controller state machine should transition following execution of any irScan that completes in the EXIT1-IR state.

irPostpend(javaScanBitIf) - The irPostpend method specifies those bits that should be shifted in on TDI after those bits specified in irScan method calls.

irPrepend(javaScanBitIf) - The irPrepend method specifies those bits that should be shifted in on TDI before those bits specified in irScan method calls.

irScan(javaScanBitIf, javaScanBitIf, byte, byte) - The irScan method uses overloading to define various manners in which to drive the TAP controller state machine to the SHIFT-DR state. The variety of different operations includes simply shifting data in, indicating expected data to be shifted out, allowing shifting of blocks of data, skipping transitions through Run Test Idle on completion. All of these different operations are usable alone, or in combination through the power of overloading.

scanAsyncReset() - The scanAsyncReset method performs an asynchronous TAP reset.

scanState(byte) - The scanState method transitions the TAP controller state machine to the state indicated by the parameter.

scanSyncReset() - The scanSyncReset method performs a synchronous TAP reset.

setTCKFrequency(int) - The setTCKFrequency method is used to set the operating frequency of TCK.

waitTCK(int) - The waitTCK method specifies the number of TCK pulses to execute.

waitTime(int) - The waitTime method specifies the time to idle in the current TAP controller state machine state.

1.6.2 Data Compression

Typically, the programming or test data, if represented in ASCII, HEX or BINARY can get overwhelmingly large. This is especially true if many devices are being programmed or tested. The data producer, however, best selects the compression algorithm used. Therefore, each data producer would select a fitting technique for data compression that provides best results for their variety of data. By not enforcing an algorithm that favors one style of data over another, you can avoid equally suboptimal results for all data. Each Java scanlet, though, must provide its own decompression algorithm.

To allow the wide variety of algorithms, the **javaScanBitIf** interface and its associated methods standardize data access techniques.

1.6.3 Java Native Interface Requirements

Eventually the TAP operations described by the **javaScanOperations** class needs to be applied to the system as electrical stimuli. The application port might be a group of processor pins in an embedded processor, a PC parallel port, a workstation serial port, a computer USB or FireWire port or a custom hardware proprietary port.

The classic manner to interface to a wide variety of disparate devices is to define a standard device interface and then supply suitable drivers for each device. In Java, this is best set up as a Java Native Interface (JNI). Having such a JNI-based API allows users full algorithmic portability. Those who run on an embedded processor use pins of the microprocessor to produce the TAP signals. If the scanlet is executed on a PC then a cable connected to the parallel port may be used to produce the TAP signals. Then again, if the scanlet is executed on automatic test equipment some complex proprietary hardware may be driving the TAP pins. All of these native calls are encapsulated in an object called the **javaScanHWIf.**

1.7 Java API for Boundary-Scan File Example

A simple example can help understanding the use of the Java API for Boundary-Scan. In addition, because this example is Java, some familiarity with algorithmic control flow languages will be helpful.

The sample code we will study, is a single JAPIBS application that can configure any device from the Xilinx XC9500 CPLD family. With some minor changes, you can invoke this single application from with a browser as an apple. You could also integrate it into a larger application to perform configuration as needed.

The application is just a regular Java application. In fact, the first section of the application is unsurprising. It is standard program fare: variables are initialized, input is validated and methods are called. When we examine the contents of the configuration methods you will see the functionality of the JAPIBS. Even then, though, it is still just a Java application. This ordinariness is what makes the JAPIBS so powerful. If you can write a Java application, you can reuse the algorithmic objects for any device and build your own custom configuration infrastructure.

The application begins with a listing of the included Java libraries. In this case, the libraries included allow IO operations. The application targets a fuller JVM of the embedded Java family or higher.

```
import java.io.InputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileOutputStream;
import java.lang.Integer;
```

The JAPIBS libraries are included to give access to the boundary-scan functionality needed to configure the devices.

```
import com.scan.javaScanOperations;
import com.scan.javaScanHWIf;
import com.scan.javaScanBitIf;
import com.scan.javaScanState;
```

The device-specific data libraries are included to allow the application to interpret the configuration data properly. In this case, the application supports data compression using Huffman encoding. As previously pointed out, the data interpretation implementation is defined to suit the target device. Some devices may opt for no compression, others may choose proprietary techniques.

```
import com.xilinx.compression.huffmanStream;
```

```
import com.xilinx.utilities.universalIO;
import xilinxCpldBits;
```

Every Java application is encapsulated in a class. In this case, the class is named xc9500cls. Other devices that use identical configuration algorithms can derive from this class and reuse what exists.

```
public class xc9500cls
{
```

It is useful to keep track of the algorithm version set up in this class.

```
static final byte VERSION = 4;
```

You must instantiate the basic JAPIBS class to get access to the boundary-scan operations.

```
private static javaScanOperations javaScanObj;
```

Then, all the local variables needed to carry out the configuration algorithm need to be defined and initialized. These include instructions for all the device operations.

```
private static xilinxCpldBits idcode ;
private static xilinxCpldBits bulk ;
private static xilinxCpldBits iscenable ;
private static xilinxCpldBits program ;
private static xilinxCpldBits verify ;
private static xilinxCpldBits bypass ;
```

Good algorithm practice includes testing the instruction register capture bits and the IDCODE value with each operation. The actual and expected values are stored in these variables.

```
private static xilinxCpldBits IRcapture;
private static xilinxCpldBits IRcaptured;
private static xilinxCpldBits deviceIDInput;
private static xilinxCpldBits deviceIDOutput;
```

Since this application will be used for all devices in the family, the IDCODE value for each family member needs to be known.

```
private static xilinxCpldBits xc9536DeviceID;
private static xilinxCpldBits xc9572DeviceID;
private static xilinxCpldBits xc95108DeviceID;
private static xilinxCpldBits xc95144DeviceID;
private static xilinxCpldBits xc95216DeviceID;
private static xilinxCpldBits xc95288DeviceID;
private static xilinxCpldBits thisDeviceID;
```

The device included a series of configuration registers to load data. The registers also capture information that is needed for the algorithm flow. In this section, we define the registers conceptually as input and output registers. Since the XC9500 family has a non-deterministic programming algorithm, some operations may need to be retries to complete successfully. A separate register is defined to store the data to be reapplied to the device. That is the retryRegister.

```
private static xilinxCpldBits iscRegister;
private static xilinxCpldBits ispVRegister;

private static xilinxCpldBits configurationRegister;
private static xilinxCpldBits resultRegister;
private static xilinxCpldBits retryRegister;
```

The TAP state is stored and tracked in the scanState object. This is intialized here.

```
private static javaScanState scanState;
```

Finally, the input file access object is defined here. This allows you to read the configuration data supplied in a separate file. The application accepts either compressed or uncompressed configuration data.

```
private static InputStream inputData1;
private static InputStream inputData;
```

The program main defines all the functionality deliverable by this application. The main will read the command line and decide from the arguments provided what the user wants to do. In this simple application, the user can select to erase, program or verify the device. In addition, as pointed out previously, configuration data in normal or compressed format is provided.

```
public static void main(String args[])
{

    byte  eraseFlag = 0, programFlag  = 0, verifyFlag = 0;
```

Using good practice, the application displays usage information if the arguments seem wrong.

```
    if (args.length < 2) {
        System.out.print("Usage:    xc9500cls    [-erase|-program|-
verify] <data file>\n");
        System.exit(-1);
    }
```

The program parses the arguments to decide what to do based on the flags supplied.

```
for (int i = 0; i < args.length; i++) {
    if (args[i].equalsIgnoreCase("-erase")) {
        eraseFlag = 1;
    } else {
        if (args[i].equalsIgnoreCase("-program")) {
            programFlag = 1;
        } else {
            if (args[i].equalsIgnoreCase("-verify")) {
                verifyFlag =1;
            } else {
                dataFile = args[i];
            }
        }
    }
}
```

Now open the data file to allow access to its contents. If the file is not found, an error message is printed and the applications stops.

```
try {
    //Create an inputStream to handle both urls and files.
    universalIO x = new universalIO(dataFile);
    if(dataFile.endsWith(".pack")){
            inputData1 = x.getInputStream();
            inputData = new huffmanStream(inputData1);
```

```
        }
        else
                inputData = x.getInputStream();
    } catch (FileNotFoundException e) {
        System.out.print("File not found!\n");
        System.exit(-1);
    }
```

After finding out there is something useful to do, initialize the support data and perform the correct operations. The initialize method sets up all the data you need to run the algorithm. It also does some simple tests to make certain the boundary-scan connections are correct and the expected device is present.

```
System.out.println("Initializing device... ");
byte device = xc9500cls.initialize( inputData );
if (device == (byte) -1) {
    System.out.print("Initialization error!\n");
    xc9500cls.terminate();
    System.exit(-1);
}
```

If the "-erase" flag was set then erase the device.

```
if (eraseFlag != 0) {
    System.out.print("Erasing device ");
    xc9500cls.erase( device );
    System.out.println("...done");
}
```

If the "-program" flag was set then program the device.

```
if (programFlag != 0) {
    System.out.print("Programming device ");
    xc9500cls.program( inputData );
    System.out.println("...done.");
}
```

Close the configuration data file since programming is complete.

```
try {
    inputData.close();
```

```
    } catch (IOException e) {
        System.out.print("File close failed!\n");
        xc9500cls.terminate();
        System.exit(-1);
    }
```

If the "-verify" flag was set then reopen the configuration data file to get back to the beginning of the file. Then call the verify method.

```
if (verifyFlag != 0) {
    try {
        universalIO y = new universalIO(dataFile);
if(args[dataFile.endsWith(".pack")){
        inputData1 =  y.getInputStream();
        inputData = new huffmanStream(inputData1);
    }
    else
        inputData = y.getInputStream();
} catch (FileNotFoundException e) {
    System.out.print("File not found!\n");
    xc9500cls.terminate();
    System.exit(-1);
}

System.out.print("Verifying device ");
if (xc9500cls.verify( inputData ) != 0) {
    System.out.println("...Verify errors idenitified!");
    xc9500cls.terminate();
    System.exit(-1);
}
System.out.println("...done.");
```

When done, close the data stream and exit.

```
    try {
        inputData.close();
    } catch (IOException e) {
        System.out.print("File close failed!\n");
        xc9500cls.terminate();
        System.exit(-1);
    }
}
```

```
            xc9500cls.terminate();
        }

    System.exit(0);
}
```

This completes the main application block. As suggested previously, it looks like a normal Java application - because it is. With the next object methods, we will begin to see use of the JAPIBS to provide TAP access and control.

We begin with the initialize method. This method initializes all objects with their value for use within the algorithmic methods. This includes defining the bit patters of the in-system configuration instructions, the expected device IDCODE values and the various data registers to be used during algorithm execution.

```
public static byte initialize( InputStream input ) {
```

The basic JAPIBS class javaScanObj gives access to all the TAP operations. This is instantiated in this method for use throughout.

```
javaScanObj = new javaScanOperations( );
```

The device's basic ISC instruction patterns are initialized here. A more sophisticated application could read these from the device's BSDL file. Here they are hard-coded.

```
iscenable = new xilinxCpldBits( (byte) 0xe8 );
iscdisable = new xilinxCpldBits( (byte) 0xf0 );
bypass = new xilinxCpldBits( (byte) 0xff );
```

The expected instruction register capture value and a variable to store the value read from the device are initialized here. As with the ISC instructions, a more sophisticated application could read this information from the device's BSDL file.

```
IRcapture = new xilinxCpldBits( (byte) 0x01 );
IRcaptured = new xilinxCpldBits( (byte) 0x00 );
```

The IDCODE values for each member of the family are set. As with the instruction pattern information, a more sophisticated application could read this information from all the devices' BSDL files.

```
xc9536DeviceID = new xilinxCpldBits( (int) 0x09502093 );
xc9572DeviceID = new xilinxCpldBits( (int) 0x09504093 );
xc95108DeviceID = new xilinxCpldBits( (int) 0x09506093 );
xc95144DeviceID = new xilinxCpldBits( (int) 0x09508093 );
xc95216DeviceID = new xilinxCpldBits( (int) 0x09512093 );
xc95288DeviceID = new xilinxCpldBits( (int) 0x09516093 );
```

The device's data registers are sized and initialized.

```
configurationRegister = new xilinxCpldBits( (int) 0x0, 27 );
resultRegister = new xilinxCpldBits( (int) 0x0, 27 );
retryRegister = new xilinxCpldBits( (int) 0x0, 27 );
```

The TAP state is initialized and the end-of-shift transition state is set for both Shift IR and Shift DR. In both cases, the state is set to Run Test/Idle.

```
scanState = new javaScanState();

javaScanObj.irEnd( javaScanState.RUN_TEST_IDLE );
javaScanObj.drEnd( javaScanState.RUN_TEST_IDLE );
```

And now, finally, some TAP operations. The algorithm begins with a synchronous transition to Test Logic Reset. The TAP controller is instructed to hold TMS high for five TCK pulses. Once in Test Logic Reset, the TAP controller is directed to transition to Run Test/Idle.

```
javaScanObj.scanSyncReset();
javaScanObj.scanState( javaScanState.RUN_TEST_IDLE );
```

Once in Run Test/Idle, shift in the BYPASS instruction and look at the bits shifted out of the device as stored in the IRcaptured variable. Then test and see if what was returned from the device equals the expected value as stored in IRcapture. If the returned value differs from the expected value then exit the operation with an error status.

```
javaScanObj.irScan( bypass, IRcaptured );
if (IRcaptured.equals( IRcapture ) != 0) {
```

```
            System.out.print("Boundary-scan shift path has open
connections.\n");
            return( (byte) -1 );
        }
```

In this case, the programming data file has the target device stored in its first four bytes. By reading out this data, the application knows what device it should be talking to and can test to make certain that it is the case. In addition, for this family, certain device-specific data must be loaded at configuration time. This value is loaded as part of configuration algorithm initialization.

```
int bytes = 0;
byte data[] = new byte[4];
try {
    bytes = inputData.read( data );
} catch (IOException e ) {
    System.out.print("IO Error!\n");
}
byte device = data[0];

switch (device) {
case 4:
    iscRegister = new xilinxCpldBits( (byte) 0x0f, 6 );
    ispVRegister = new xilinxCpldBits( (byte) 0x07, 6 );
    thisDeviceID = xc9536DeviceID;
    break;
case 8:
    iscRegister = new xilinxCpldBits( (byte) 0x3f, 8 );
    ispVRegister = new xilinxCpldBits( (byte) 0x1f, 8 );
    thisDeviceID = xc9572DeviceID;
    break;
case 12:
    iscRegister = new xilinxCpldBits( 0x0ff, 10 );
    ispVRegister = new xilinxCpldBits( 0x07f, 10 );
    thisDeviceID = xc95108DeviceID;
    break;
case 16:
    iscRegister = new xilinxCpldBits( 0x3ff, 12 );
    ispVRegister = new xilinxCpldBits( 0x1ff, 12 );
    thisDeviceID = xc95144DeviceID;
    break;
```

```
        case 24:
            iscRegister = new xilinxCpldBits( 0x3fff, 16 );
            ispVRegister = new xilinxCpldBits( 0x1fff, 16 );
            thisDeviceID = xc95216DeviceID;
            break;
        case 32:
            iscRegister = new xilinxCpldBits( 0x3ffff, 20 );
            ispVRegister = new xilinxCpldBits( 0x1ffff, 20 );
            thisDeviceID = xc95288DeviceID;
            break;
        default:
            return((byte) -1);
        }
```

The IDCODE value is then read out of the device and compared against the expected value.

```
        deviceIDOutput = getIDCODE();
        if (deviceIDOutput.equals( thisDeviceID, 27 ) != 0) {
            System.out.print("Device ID check failed\n");
            return((byte) -1);
        }
        return( device );
    }
```

That is the end of the initialize method. It returns a coded value of the device for use in the algorithmic steps that follow.

The getIDCODE method loads the IDCODE instruction and reads the device's IDCODE value and returns the value read from the device to the calling program.

```
    public static xilinxCpldBits getIDCODE( ) {
```

The IDCODE instruction bit pattern is defined. Then a sequence of ones is defined to shift into the device to get the IDCODE value out. Finally, a variable is defined to hold the device's IDCODE value (deviceID).

```
        idcode = new xilinxCpldBits( (byte) 0xfe );
        deviceIDInput = new xilinxCpldBits( (int) 0xffffffff );
        xilinxCpldBits deviceID = new xilinxCpldBits( (int) 0xffffffff );
```

The steps in reading the IDCODE involve, first shifting in the IDCODE instruction…

```
javaScanObj.irScan( idcode );
```

…then shifting in the sequence of ones to shift out the device's IDCODE value…

```
javaScanObj.drScan( deviceIDInput, deviceID );
```

…which is returned to the calling program for further processing.

```
        return( deviceID );
    }
```

The next method defines the erase algorithm.

```
public static void erase( byte device ) {
```

The wait time associated with the erase operation is defined as 1,300,000 microseconds.

```
int wait_time = 1300000;
```

The erase instruction bit pattern is defined here. As suggested previously, a more sophisticated application could read this information from the device's BSDL file.

```
bulk = new xilinxCpldBits( (byte) 0xed );
```

Now we begin execution of the erase algorithm. First the ISC_ENABLE instruction is loaded to put the device in in-system configuration mode. Then the associated data register value as defined by iscRegister is shifted in.

```
javaScanObj.irScan( iscenable );
javaScanObj.drScan( iscRegister );
```

Then the variables used to store the input values and the values shifted out of the device are cleared.

```
configurationRegister.clear();
```

```
resultRegister.clear();
```

Now we shift in the erase instruction (named bulk). Then the data to be shifted in is defined as required by the algorithm. The data bits 0 and 1 define the incoming status that signals valid data is shifted in. Data bits 5 through 21 are the sector address which is zero. Data bits 2 through 7 are don't care bits for erase and are set to all ones. The data is then shifted into the device. Since the end state of the TAP is defined to be Run Test/Idle, the specified wait is performed in that state, as needed.

```
javaScanObj.irScan( bulk );
configurationRegister.setBits( 0, 2, (byte) 0x2 );
configurationRegister.setBits( 22, 5, (byte) 0x0 );
configurationRegister.setBits( 2, 8, (byte) 0xff );
javaScanObj.drScan( configurationRegister );
javaScanObj.waitTime( wait_time );
```

This device has a non-deterministic configuration algorithm. The device responds with a status that signals if an extra try is needed to complete the operation. In addition, the next sector address is used to shift out the result data. This involves setting the input data register bits 5 through 21 to 1.

```
configurationRegister.setBits( 22, 5, (byte) 0x1 );
xilinxCpldBits repeat = new xilinxCpldBits( (byte) 0x3, 2 );
javaScanObj.drScan( configurationRegister, resultRegister );
resultRegister.getBits( 0, 2, repeat );
```

The status bits that signal if extra tries are needed are stored in the repeat variable. If the value is 3 then the operation completed successfully. Otherwise, another try is needed.

```
while( repeat.equals( (byte) 0x3 ) != 0) {
    javaScanObj.waitTime( wait_time );
    resultRegister.setBits( 0, 2, (byte) 0x2 );
    javaScanObj.drScan( resultRegister, resultRegister );
    resultRegister.getBits( 0, 2, repeat );
}
```

The erase algorithm requires addressing two distinct spaces in the device. In this phase, the second sector is erased using a method identical with that of the first sector. The only difference is the sector address changes (to 1, as previously set).

```
javaScanObj.drScan( configurationRegister );
javaScanObj.waitTime( wait_time );
javaScanObj.drScan( configurationRegister, resultRegister );
resultRegister.getBits( 0, 2, repeat );
while( repeat.equals( (byte) 0x3 ) != 0) {
        javaScanObj.waitTime( wait_time );
        resultRegister.setBits( 0, 2, (byte) 0x2 );
        javaScanObj.drScan( resultRegister, resultRegister );
        resultRegister.getBits( 0, 2, repeat );
}
```

The final operation is to leave in-system configuration mode by loading the ISC_DISABLE instruction

```
    javaScanObj.irScan( iscdisable );
}
```

The next method defines the programming algorithm. It is identical with the erase algorithm in flow. The program instruction is loaded, the program data is shifted in, the operation completes in the Run Test/Idle state, the device status is tested and extra tries are carried out as needed.

```
public static byte program( InputStream inputData ) {
    int bytes = 0;
    byte error;
    byte data[] = new byte[4];
```

The program instruction bit pattern is defined here. As pointed out previously, a more sophisticated application could read this information from the device's BSDL file.

```
program = new xilinxCpldBits( (byte) 0xea );
```

Now we begin execution of the program algorithm. First the ISC_ENABLE instruction is loaded to put the device in in-system configuration mode. Then the associated data register value as defined by iscRegister is shifted in.

```
javaScanObj.irScan( iscenable );
javaScanObj.drScan( iscRegister );
```

Now we shift in the program instruction (named program). Then the data to be shifted in is read from the input data file. The data bits 0 and 1 define the incoming status that signals valid data is shifted in. Data bits 2 through 24 are the address and configuration data as read from the file. The data is then shifted into the device. Since the end state of the TAP is defined to be Run Test/Idle, the specified wait is completed in that state, as needed.

```
javaScanObj.irScan( program );
xilinxCpldBits repeat = new xilinxCpldBits( (byte) 0x3, 2 );
```

The try operation is used in Java to trap IO errors when reading files. If a file access fails then the code associated with the catch operation below is carried out.

```
try {
```

The first time through the operation, no data needs to be shifted out. The first variable says if this is the first time or not.

```
byte first = 1;
```

Read four bytes of data from the configuration data file.

```
while( (bytes = inputData.read( data )) != -1) {
```

Clear the contents of the variable used to store the data to be shifted into the device (configurationRegister).

```
configurationRegister.clear();
```

The data bits 0 and 1 define the incoming status that signals valid data is shifted in. Data bits 2 through 24 are the configuration address and data. This is the information read from the configuration data file. The stored data is then shifted into the device. Since the end state of the TAP is defined to be Run Test/Idle, the specified wait is completed in that state, as needed.

```
configurationRegister.setBits( 0, 2, (byte) 0x2 );
configurationRegister.setBits( 2, 25, data );
javaScanObj.drScan(configurationRegister,resultRegister);
javaScanObj.waitTime( 640 );
```

If this is the first time through then the status will not be checked. Otherwise, the device status is available in the resultRegister variable.

```
if (first == 0) {
```

The device status is collected in the repeat variable.

```
resultRegister.getBits( 0, 2, repeat );
```

If the status is equal to 3 then the program operation completed successfully. Otherwise, the operation needs to be repeated for the address and data just shifted in.

```
while( repeat.equals( (byte) 0x3 ) != 0) {
  javaScanObj.drScan( retryRegister );
```

The wait time for the program operation is 640 microseconds. As with the erase, it completes in Run Test/Idle. Since the end state of the drScan is Run Test/Idle, the TAP is already in that state.

```
  javaScanObj.waitTime( 640 );
  javaScanObj.drScan(retryRegister, resultRegister );
  resultRegister.getBits( 0, 2, repeat );
}
```

Since we have completed the first pass through the algorithm, we set the first flag to zero.

```
  first = 0;
}
```

Just in case a retry needs to be done, save the data to shifted in again in the retryRegister variable.

```
    retryRegister.copy( configurationRegister );
  }
} catch (IOException e ) {
  System.out.print("IO Error!\n");
}
```

When you reach the last address to be configured, the configurationRegister variable contains that final data. Shift in the value to

be configured and collect the result of the previous program operation in resultRegister. Get the status bits in the repeat variable and test if the return status is 3.

```
javaScanObj.drScan( configurationRegister, resultRegister );
resultRegister.getBits( 0, 2, repeat );
while( repeat.equals( (byte) 0x3 ) != 0) {
```

If the return status did not signal success, the retry value is already loaded so you only need to wait in Run Test/Idle for the program operation to complete. Then shift in the value again and test the status until you read success status.

```
    javaScanObj.waitTime( 640 );
    javaScanObj.drScan(configurationRegister,resultRegister );
    resultRegister.getBits( 0, 2, repeat );
}
```

After the final address is programmed, load the ISC_DISABLE instruction to exit ISC mode.

```
javaScanObj.irScan( iscdisable );
return(0);
}
```

The next method describes the device configuration verification algorithm. This algorithm flow is identical with the program method but does not need wait times in Run Test/Idle or retries since the read operation is deterministic.

```
public static byte verify( InputStream inputData ) {

int bytes = 0;
byte data[] = new byte[4];
byte actualData = 0x0, expectedData = 0x0;
```

The verify instruction bit pattern is defined here. As suggested previously, a more sophisticated application could read this information from the device's BSDL file.

```
verify = new xilinxCpldBits( (byte) 0xee );
```

The first four bytes of the configuration data file are the device information. Since that data is not used as part of the verify operation it is read and discarded.

```
try {
   bytes = inputData.read( data );
} catch (IOException e ) {
   System.out.print("IO Error!\n");
}
```

Now we begin execution of the program algorithm. First the ISC_ENABLE instruction is loaded to put the device in in-system configuration mode. Then the associated data register value as defined by iscVRegister is shifted in.

```
javaScanObj.irScan( iscenable );
javaScanObj.drScan( ispVRegister );
```

Now we shift in the verify instruction (named verify). Then the data to be shifted in is read from the input data file. The data bits 0 and 1 define the incoming status that signals valid data is shifted in. Data bits 2 through 24 are the address and configuration data as read from the file. The only information read by the device is the address. Having the data will be useful to compare against what was returned from the device. The data read is then shifted into the device.

```
javaScanObj.irScan( verify );
try {
```

As with the program algorithm, the first time through, no data can be shifted out. The first shift sets the address from which to read and only after it is completed can valid data be read out of the device. The first variable tells if this is the first time or not.

```
byte first = 1;
while( (bytes = inputData.read( data )) != -1) {
```

Clear the contents of the variable used to store the data to be shifted into the device (configurationRegister).

```
configurationRegister.clear();
```

The data bits 0 and 1 define the incoming status that signals valid data is shifted in. Data bits 2 through 24 are the configuration address and data. This is the information read from the configuration data file. This stored data is then shifted into the device.

```
configurationRegister.setBits( 0, 2, (byte) 0x2 );
configurationRegister.setBits( 2, 25, data );
javaScanObj.drScan(configurationRegister,resultRegister);
```

If this is not the first time the data read from the resultRegister will be valid. The bits 2 through 7 are the configuration data read from the device. These are compared against the expected value. If they differ, an error is signaled and the method exits.

```
if (first == 0) {
   actualData = resultRegister.getByte( 2, 8 );
   if (actualData != expectedData) {
   System.out.println("Verification error");
   return(-1);
   }
} else {
```

If it is the first time through do nothing but set the flag to indicate the first time has been completed.

```
   first = 0;
}
```

Set up the expected data by reading the bits 2 through 7 from the configurationRegister variable.

```
      expectedData = configurationRegister.getByte( 2, 8 );
   }
} catch (IOException e ) {
   System.out.print("IO Error!\n");
}
```

Read out and check the value of the last configuration word.

```
javaScanObj.drScan( configurationRegister, resultRegister );
actualData = resultRegister.getByte( 2, 8 );
if (actualData != expectedData) {
```

```
            System.out.println("Verification error");
            return(-1);
        }
```

After the final address is read, load the ISC_DISABLE instruction to exit ISC mode.

```
        javaScanObj.irScan( iscdisable );
        return(0);
    }
```

The terminate method loads the ISC_DISABLE instruction to exit ISC mode and then loads the BYPASS instruction to complete the transition out of ISC mode and enable the programmed function of the device.

```
    public static void terminate() {
        javaScanObj.irScan( iscdisable );
        javaScanObj.irScan( bypass );
        System.out.print("Completed...\n" );
    }

}
```

If you are used to application programming then the JAPIBS merely provides a set of building blocks with which to develop boundary-scan applications of any sort. It has many functions that make it well suited to configuration algorithm description.

The JAPIBS application can be as simple or complex as needed. The sample application, for instance, supports only single device chains. An adapted version of the initialize method could be developed to supply any pre- and post-padding if instruction and data register bits to support the XC9500 devices in an arbitrarily long boundary-scan chain.

A sophisticated systems designer has the freedom to build her own boundary-scan applications and integrate them into the broader system application. Less demanding designers can use JAPIBS-based applications as building blocks for simple device configuration applications.

1.8 Using the Java API for Boundary-Scan

By building on the experience of SVF and STAPL and basing the approach on an existing programming language, JAPIBS provides a more complete system solution. It leverages Java's adaptability, portability and integrability to ease incorporation of configuration to your system software solution. In addition, through use of Java's extension networking and security libraries, it is straightforward to deploy your configuration functionality on the Internet securely.

Separating configuration data from the algorithm makes updating either the algorithm or the data independently much simpler.

The target platforms are those for which a Java Virtual Machine (JVM) is available. As with SVF, a key need was developing a format easily produced by test software that was usable on a multiplicity of test platforms. Care must be taken when developing JAPIBS scanlets to ensure the Java libraries used are supported across the space of JVMs targeted. For instance, if you are targeting the Java Card JVM then you will not be able to use the Java networking libraries. If however, you are targeting Embedded Java and Enterprise Java you will be able to use these libraries.

Identifying a JVM for your platform of choice may be a challenge. JVMs are widely available for Windows, Solaris and Linux. The availability of JVMs for embedded systems, however, is more limited. Major RTOS manufacturers do supply JVMs as add-ons to their RTOSs. If, however, you develop your own embedded system infrastructure, you will have to find your own JVM and customize it for your system. While open source JVMs exist (like Japhar), the effort in customizing it may be significant. The value of the customization effort of JVM will be increased if the configuration functionality is essential to system operation.

The vendor software produces a basic scanlet but it will typically assume a Java Card JVM and constrain its operations to the most portable language and library subset. This suggests that JAPIBS scanlets may need to be customized by the system designer to achieve the functionality required. Added functionality, however, will limit the global portability.

Because different scanlets may use different data access and compression algorithms, the system designer needs to make certain the needed interface implementations (provided by the vendor) are included in the run time environment.

The system designer has a fair amount of responsibility in building a system to support a scanlet. The good news though is the responsibility is identical with that of installing Java in the chosen run time environment. If the system already uses Java then the effort is slight. If the system does not use Java then the mechanisms are well documented with a rich variety of tutorial and expert knowledge. In the end, this effort is similar to other approaches but the solutions are already available unlike SVF or STAPL in which the onus is on the system designer to discover the correct approach.

Chapter 7

CONFIGURATION SPECIFICATION AND DESCRIPTION LANGUAGES

IEEE Standard 1532

1. IEEE STD 1532 BSDL

In 1996, there was a series of informal industry discussions about the Babel-like status of in-system configurable devices and languages. This led to the organization, by Agilent (then still part of Hewlett Packard) and Xilinx, of a programmable logic summit to explore the possible standardization of both the configuration behavior of programmable devices and their description.

What was clear from the start was most devices were using the basic communication protocol and associated state machine of IEEE STD 1149.1. (IEEE STD 1149.1 is also known as JTAG or boundary-scan, but experts in the field will be quick to tell you that these common synonyms are subtly different from IEEE STD 1149.1). The end user community demanding simultaneous support of in-system configuration and boundary-scan interconnect test drove this. Since IEEE STD 1149.1 was designed to allow for arbitrary extension of the instruction set to support other test or non test-related operations, as needed, it made sense to attach configuration functionality onto the IEEE STD 1149.1 infrastructure.

After the summit meeting, there was broad general agreement to continue toward standardization. There was agreement that leveraging the existing IEEE STD 1149.1 infrastructure and knowledge base would benefit in-system configuration device, tool, system and application development. In addition, such standardization also promised the possibility of multi-vendor concurrent programming. This would allow end users to choose programmable devices according to their design needs and still be able to minimize system configuration time, by increasing overall throughput of

systems in manufacturing. We will examine the benefits of concurrent programming in a later section.

After a prolonged and thorough definition process, the decision was that standardization process needed agreement on two separate but related elements. The first was to define a standardized hardware behavior. That is, devices that would claim to adhere to the standard would be needed to follow strict rules governing the externally observable behavior of the device before, during and after configuration. In addition, a set of rules would govern the device's use of the IEEE STD 1149.1 TAP states restricting what can be done in each state and what state trajectories should be allowed and used. We will deal more with the specific hardware rules and behaviors in a later section.

The second point of agreement was that to promote use of IEEE STD 1532 compliant devices, some algorithmic description would be needed. Once again, by turning to the IEEE STD 1149.1 infrastructure, it was decided that Boundary-Scan Description Language (BSDL) would be the best method to describe the necessary operations. BSDL already had the idea of extension in place. A BSDL extension is a construct for adding application-specific information to a BSDL file. BSDL parsers that don't understand the contents of the extension skip over it. BSDL parsers that do understand the extension, parse and interpret it.

1.1 Basic IEEE STD 1532 BSDL File Structure

BSDL is a subset of VHDL (IEEE STD 1076-1993). However, BSDL is not necessarily 100% VHDL compliant. You cannot (nor should you have the need to) execute a BSDL description as a standard VHDL file. The user should be aware that some modification of BSDL files may be needed should they be used as input to VHDL-based tools. No way has been found of avoiding this small amount of effort without introducing further undesirable complications. Specifically, the BSDL use statement may need editing because of tool and file system dependencies. Syntax of the statements, as defined, is legal VHDL; however, an added prefix (identifying a library in which the Standard VHDL Package will be found) must be added for some applications. A syntax lacking such a prefix has been chosen to force an error in such an application rather than risk unpredictable and confusing errors because of including an inappropriate prefix.

It should also be noted that BSDL does not employ all the syntactic elements of VHDL. Only those elements needed to meet the scope of BSDL

are used. Sometimes, only a subset of a particular VHDL language element syntax is used in BSDL.

Further, for cases in which a feature could be described in several ways within VHDL, a restricted set of ways has been selected and defined exactly as the standard practice for BSDL. This restriction simplifies the application of the VHDL subset for BSDL, particularly for tools that are only required to read or produce BSDL (that is, tools that have no requirement to read or write the full VHDL language).

In addition, BSDL imposes extra requirements on the syntax and content of certain character strings— that is, sequences of characters between quotation marks (for example, "EXTEST"). A VHDL parser will not check the information in these strings. In contrast, a BSDL parser shall check that the information in the strings is suitable for the relevant parameters or attributes for which such strings might be values.

Most of the BSDL file is devoted to describing IEEE STD 1149.1 boundary-scan capabilities of a device. This is well described in both IEEE STD 1149.1 and other textbooks on the matter. Therefore, details on the test sections will not be covered here. Instead we will focus on the sections of a standard BSDL file relevant to IEEE STD 1532 and on the IEEE STD 1532 BSDL extension.

1.1.1 IEEE STD 1149.1 BSDL Attributes

Because IEEE STD 1532 is built on the foundation of IEEE STD 1149.1, they share the same basic BSDL. IEEE STD 1532 does however, make certain IEEE STD 1149.1 optional attributes compulsory. This was mandated owing the utility of these functions to the needs of in-system configuration-based applications.

The two such attributes mandated are IDCODE_REGISTER and USERCODE_REGISTER. Since both the IDCODE and USERCODE instructions are mandated by IEEE STD 1532 (rather than being optional as in IEEE STD 1149.1) these attributes, that show the resultant data values associated with these instructions, must be specified.

The IDCODE instruction allows electrical identification of the devices on the boundary-scan chain. The USERCODE instruction allows electrical identification of the programmed contents of the devices. These two instructions together allow complete identification of the connected system.

This can promote remote field update when physical access to identify the system contents visually is not possible.

Sample specifications of these attributes follow below:

```
attribute IDCODE_REGISTER of A_Device: entity is
"0011" & -- Version
"1111001011010000" &      -- Part number
"00101010101" &           -- Identity of the manufacturer
"1";                      -- Required by IEEE STD 1149.1-1990
```

for IDCODE_REGISTER. This represents the value shifted out when in Shift_DR and the IDCODE instruction is active.

The example below:

```
attribute USERCODE_REGISTER of A_Device: entity is
"10XX0000011001111" &     -- Start 1st 32-bit pattern
"0000000000001X11,"&      -- End 1st 32-bit pattern
"XXXX0000010011000" &     -- Start 2nd 32-bit pattern
"0000111110011000";       -- End 2nd 32-bit pattern
```

for USERCODE_REGISTER represents the value shifted out when in Shift_DR and the USERCODE instruction is active.

1.1.2 The ISC_Pin_Behavior Attribute

There are two choices for system IO pin behavior while the device is being configured. Either the pins can float or they can be clamped and held specific user preloaded values. This attribute is made available to show to the controlling software (and the end user) which behavior is supported by this device.

Samples of this attribute are as follows:

```
attribute ISC_Pin_Behavior of A_Nother_Device:entity is "CLAMP";
```

specifies that device pins have the clamping behavior and:

```
attribute ISC_Pin_Behavior of One_More_Device:entity is "HIGHZ";
```

specifies that device pins float while the device is being configured.

When a device is being programmed, it is valuable to drive the pins of the device being programmed to state that ensure that system remains quiescent. This might include setting active device control and enable pins to ensure these devices are idle during configuration. Failing that, knowing the connected pins will float allows designer to design their systems with pull-ups or pull-downs on the proper wires will ensure correct and safe system state during configuration. This attribute tells the designer what to do to use this device properly.

1.1.3 The ISC_Fixed_System_Pins Attribute

The standard allows there might exist devices in which not all devices pins are configurable. You could imagine a device as pictured in Figure 7-1 that consists of both configurable sections and a fixed function core. Further, it is reasonable to assume the fixed function core has some IOs that are pinned out. In that situation, the core's IOs are not required to display the same behavior as the configurable sections IOs while the device is being configured. In fact, it would probably be desirable to have this flexibility. If the core is a microprocessor, you would probably not want to be needed to stop all microprocessor operations while the configurable section is being programmed.

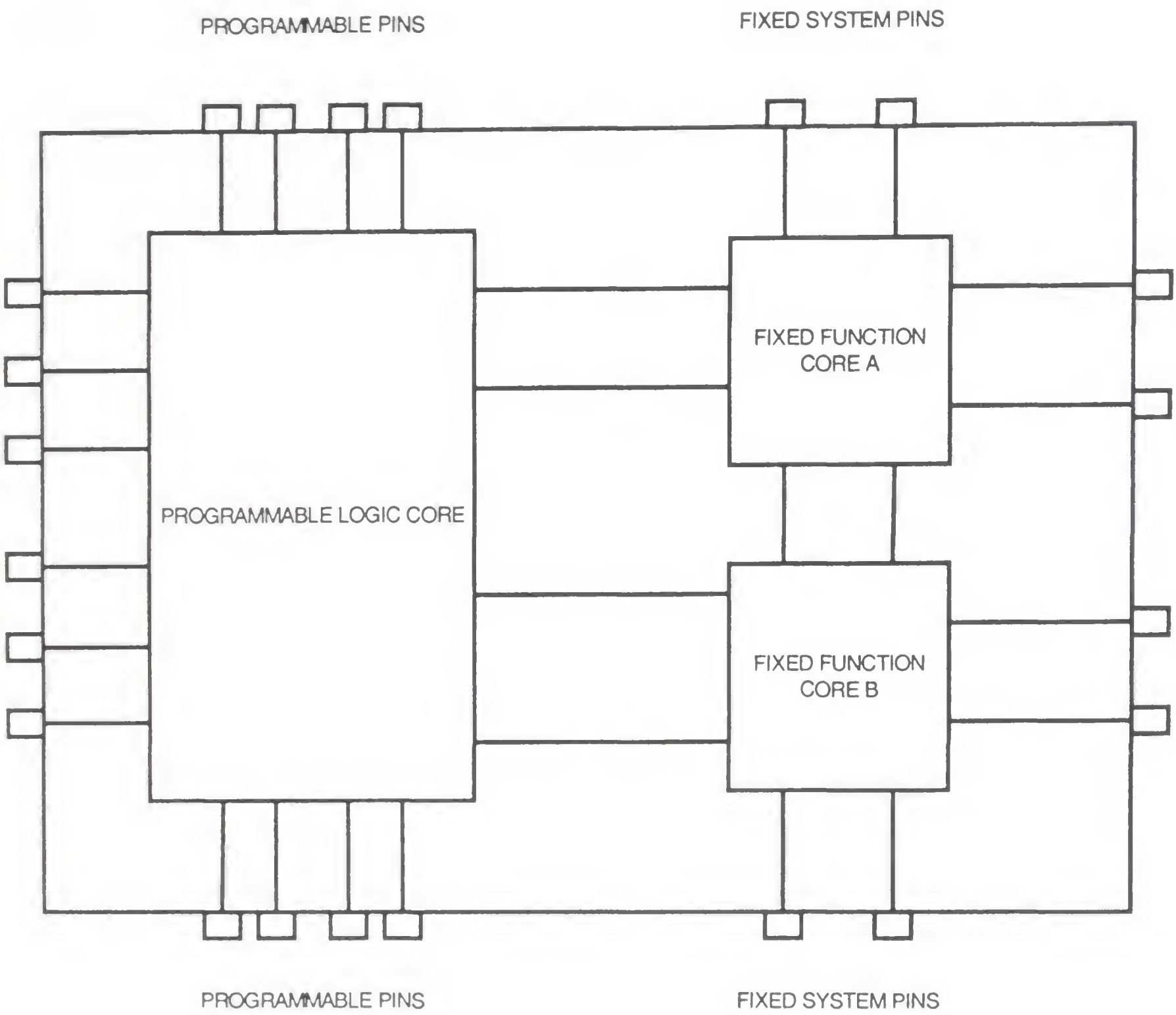

Figure 7-1. A Programmable Device with a Fixed Function Core

To show which system IO pins do not display the behavior specified by the ISC_Pin_Behavior attribute, you use the ISC_Fixed_System_pins attribute. This attribute consists of a list of pins names taken from the logical port description statement that are fully functional during ISC operations.

This example:

Attribute ISC_Fixed_System_Pins of Some_Device : entity is
`"data_bus, INIT, CS(1), sys_clock";`

shows that the pins listed are not affected by transitions into or out of configuration mode and continue to work normally.

1.1.4 The ISC_Status Attribute

IEEE STD 1532 details a specific mechanism for compliant devices to communicate the status of completed operations. This method involves providing specific data capture bits as status indicators. The standard does not mandate the use of this specific approach although it strongly recommends that some approach be used. So, although the mechanism may be preferred and support for it is built-in, other proprietary schemes can be used. Support for proprietary status schemes requires proper coding of the device algorithms specified in later attributes of the BSDL.

This attribute is used to show whether the standard status scheme is implemented in this device or not. Examples of the use of this attribute are as follows:

```
attribute ISC_Status of Device_Got_It:entity is "Implemented";
```

In the example, the device uses the standard status reporting mechanism.

```
attribute ISC_Status of Device_Dont_Got_It:entity is "Not
Implemented";
```

In the example, the device is not equipped with the standard status reporting mechanism.

1.1.5 The ISC_Blank_Usercode Attribute

The USERCODE instruction is mandatory for devices that comply with IEEE STD 1532. A valuable capability of any IEEE STD 1532 environment would be the ability to determine automatically if the device USERCODE data had already been configured. Since the logic value of unprogrammed cells depends on both the implementation technology and the whims of the IC designers, it was important to be able to specify the exact value with which a blank USERCODE would respond.

Specifying this attribute is illustrated by the following example:

```
Attribute ISC_Blank_Usercode of PLD1:entity is
"11111111111111111111111111111111";
```

1.1.6 The ISC_Security Attribute

As with device status, the standard describes one security method that, if implemented, would assure automated support in any IEEE STD 1532 compliant tool set but allows for any proprietary variations. Although no specific approach is mandated, some security mechanism is usually implemented. Typically, the basic security supplied is the ability to hinder read back of programmed data.

The method described by the standard is pictured in Figure 7-2. It includes the ability to protect the device against unwanted erasure, configuration or read back. These security mechanisms can be implemented together or separately. In addition, this method optionally allows the various protection mechanisms to be enabled by a key of specified length. The figure included shows an implementation of a key enabled version with all three security provisions.

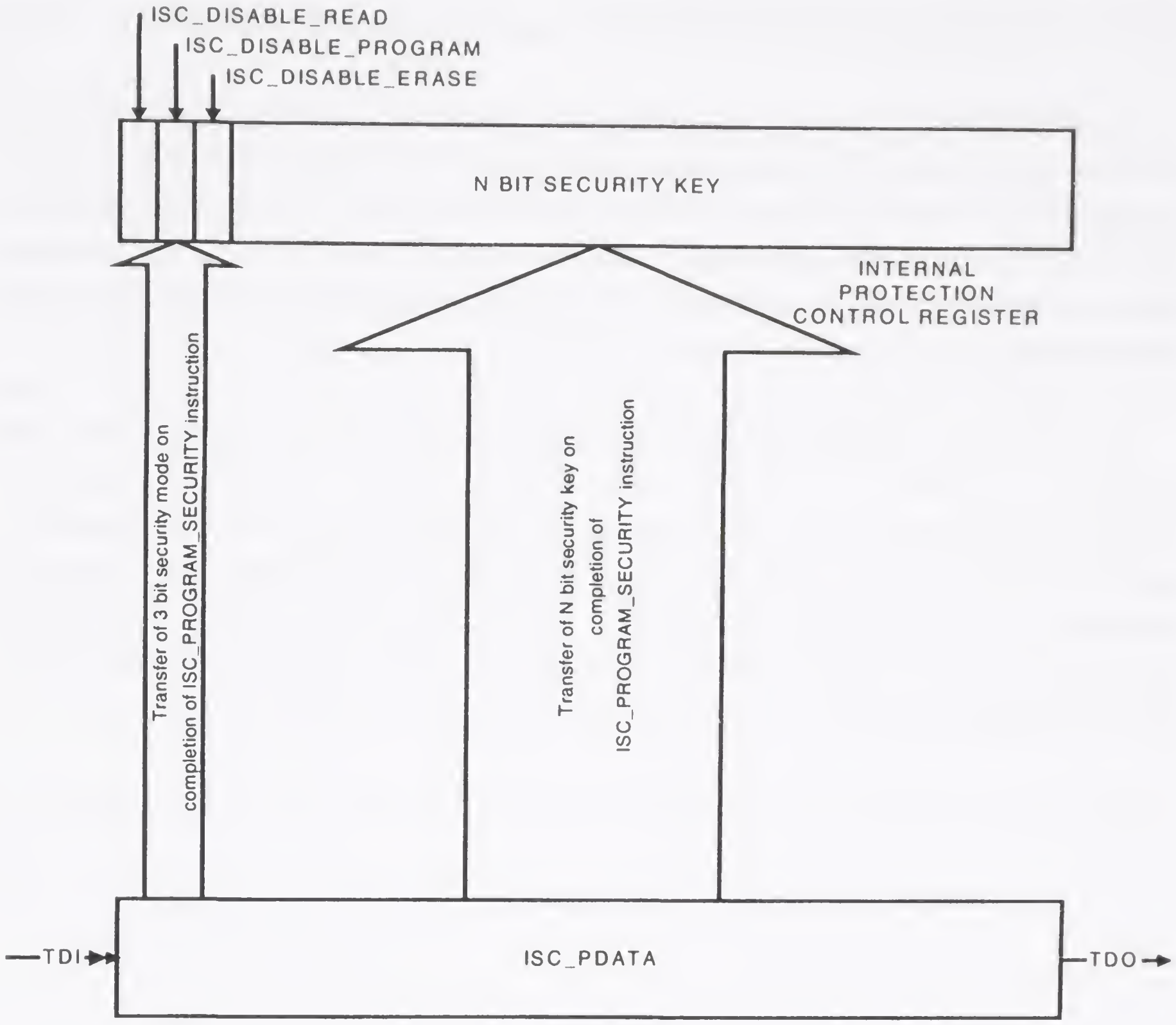

Figure 7-2. The IEEE STD 1532 ISC Security Mechanism

The standard implementation is controlled by the ISC_PROGRAM_SECURITY instruction. Security data is shifted in using the ISC_Pdata register. The data includes a key and three bits to enable or disable the protection. The protection types include means to disable unauthorized reads, programs or erases.

The security can be set and cleared only when the correct key is loaded. The key is set when the default all-zeroes key is programmed with a non-zero value. The all ones key is reserved for permanent security. Once the all ones value is programmed into the device, the security setting cannot be changed. There is no way to change the key or therefore the device security anytime in the future. A non-one key can be erased only using the erase instruction destroying not just the key but the programmed contents of the device.

The ISC_SECURITY attribute need only be specified if the security mechanism is implemented exactly as described by the standard. Proprietary

implementations call for description of the operation of the security mechanism in the flow section of the BSDL file.

The specifics of the attribute and what they mean are best indicated through example. Consider the following:

```
attribute ISC_Security of Secure_Device:entity is
"ISC_Disable_Read 31, " & -- Bit 31 controls read security
"ISC_Disable_Program 30, " & -- Bit 30 controls program security
"ISC_Disable_Erase 29, " & -- Bit 29 controls erasure security
"ISC_Disable_Key 28-0 " ; -- Bits 28 down to 0 are a key
```

The security function is controlled by the ISC_PROGRAM_SECURITY instruction that targets the ISC_PData register. The first three fields identify a bit number within ISC_PData for each of three protection signals ISC_Disable_Read, ISC_Disable_Program and ISC_Disable_Erase, if any exist. If these specified bits are asserted then the associated security mechanism is set up to be enabled.

The fourth field identifies the bits (if any exist) that form the security key. If the optional key is implemented then the need for enabling the security is that the security enabling bits are asserted and that correct key is loaded. To improve security, the key is not specified in the BSDL file. The key should be provided separately by the device manufacturer.

1.1.7 Description of ISC Algorithms in the BSDL File

The most complicated and in many ways, the most important parts of the IEEE STD 1532 BSDL extension are those parts that describe the algorithms that access the configuration memory of the ISC device. This section is broken into a hierarchy of three separate attributes that use the BSDL ISC instructions and build on one another to create a description of all the configuration operation possibilities for a single device. The three attributes are the ISC_Flow, ISC_Procedure and ISC_Action.

1.1.8 ISC_Flow

This attribute is the lowest level attribute and describes the instructions and associated data that must be loaded to carry out either a complete task or a portion of a task. The basic philosophy behind the flow is that at the atomic level, a step in an ISC operation consists of

1. Loading an instruction then,
2. Loading data associated with it then,
3. Going to Run Test/Idle and waiting and then,
4. Going back to Shift DR to shift out a result on TDO.

An ISC function is described by putting together these atomic building blocks in sequence. These operations, when strung together, can be optimized but such optimization is not needed for the ISC function to work.

An ISC function or a portion of an ISC function that is built up from these atomic operations, can generally be described by a construct that describes setting up initial conditions, looping on a series of atomic operations and then setting up some terminating conditions. It is exactly this algorithmic control flow that is described by the ISC_Flow attribute. To illustrate this, consider the following example:

```
attribute ISC_Flow of One_Example:entity is
”Program(Array) “ &
”Initialize “ &
” (ISC_ADDRESS_SHIFT 16:$addr=0 wait TCK 1) “ &
” (ISC_DATA_SHIFT 200:? wait TCK 1) “ &
” (ISC_PROGRAM wait 1.0e-2)” &
” (ISC_DISCHARGE wait 1.0e-3)” &
”Repeat 65535 “ &
” (ISC_ADDRESS_SHIFT 16:$addr+1 wait TCK 1) “ &
” (ISC_DATA_SHIFT 200:? wait TCK 1) “ &
” (ISC_PROGRAM wait 1.0e-2)” &
” (ISC_DISCHARGE wait 1.0e-3)” &
”Terminate “ &
”(ISC_ADDRESS_SHIFT 16:FFFF wait TCK 1)” &
”(ISC_DATA_SHIFT 200:0 wait TCK 1 198:0*0,2:2*3);”
```

Now let us analyze the specified ISC_Flow. After the declaration of the attribute name, the entity specifies the flow name itself. In this case, the flow name is *Program*. The specifier in parentheses indicates the name of the data block associated with this flow description. There is a separate ISC data file. This file contains the configuration (and any other) data required by the configuration algorithm. This file and its organization will be discussed more later but for now it is sufficient to note that the label *Array* identifies the data in the ISC data file to be used by the ISC_Flow.

The flow itself is broken up into Initialize, Repeat and Terminate sections. They are performed in that order and all sections are optional but at least one must exist. This means that any flow must have at least one of the Initialize, Repeat or Terminate sections.

In examining the Initialize section of our example, it should be noted that it is performed only once, in the order that it is specified. The basic fields of each parentheses-enclosed statement are broken into the following elements:

(INSTRUCTION_LOAD DATA_LOAD RTI_ACTION
DATA_CAPTURE)

In the INSTRUCTION_LOAD section, the actual instruction bit pattern that should be loaded is specified. This is accomplished by traversing the IEEE STD 1149.1 state machine to the Shift IR state and shifting in the specified instruction bit pattern in the normal manner. After that is completed, the state machine is guided to the Shift DR state skipping Run Test/Idle to load the data specified in the DATA_LOAD field. After the data has been shifted in and loaded, the state machine is guided to Run Test/Idle. There it can wait for a certain number of TCK pulses, a certain absolute amount of time or a sequential combination of the two in that order. The exact behavior is specified by the RTI_ACTION field. When that RTI_ACTION has completed, the state machine is guided to Shift_DR again to shift in don't care data for the expressed purpose of shifting out the capture data as specified in the DATA_CAPTURE field for comparison or output. The state machine is then guided back to execute the next INSTRUCTION_LOAD skipping the Run Test/Idle state. Either of the DATA_LOAD or the DATA_CAPTURE fields (or both) can be left out

Please note the sequences of these instruction and data loads might be optimized when performed. For instance, the same instruction may not be loaded multiple times, if repeated; DATA_LOADs and DATA_CAPTUREs may be interleaved to reduce the shift time. Conversely, they also might **not** be optimized. IEEE STD 1532 compliant devices must be able to operate correctly, regardless of whether the flows are optimized or not.

Now that the specifics of the constituent statements of the flow are understood, let us return to the Initialize block.

" (ISC_ADDRESS_SHIFT 16:$addr=0 wait TCK 1) " &
" (ISC_DATA_SHIFT 200:? wait TCK 1) " &

```
" (ISC_PROGRAM wait 1.0e-2)" &
" (ISC_DISCHARGE wait 1.0e-3)" &
```

The first statement indicates that the ISC_ADDRESS_SHIFT instruction bit pattern is to be loaded into the device. Then after skipping Run Test/Idle 16 bits of 0 (represented here as a hex value) are loaded while in the Shift DR state. The zero value is also stored in the local variable "addr" (variables are pointed out by the "$" prefix and have a scope within a single flow only). After the 16 0's are shifted in, the state machine is guided to Run Test/Idle for a single TCK pulse. Note that in this case there is no capture data to be examined so the shifting out step is skipped. It is noteworthy, that if there are multiple devices being accessed concurrently, one of which needs capture data then this device might carry out the capture step (but ignore the resulting data shifted out). Then the next statement is executed similarly.

The "?" indicates that the 200 bits of data required is to be read from the ISC data file. The ISC_PROGRAM and ISC_DISCHARGE instructions have no input or output data associated with them so the transitions to the Shift DR can be skipped.

This completes the execution of the Initialize block. Next, the Repeat block will be carried out. This is identical with the execution of the Initialize block with the following differences. A number appears after the Repeat keyword. This number points out the maximum number of times that all the statements in the repeat block should be carried out in order. In our example, this means the indicated block of 4 instruction and data operations should be performed no more than 65535 times. If the file supplying data to the instructions in the Repeat block (with the "?" character) is exhausted before the maximum loop count is reached, the repeat block finishes without error (there must however, be enough data in the file to complete executing all the instructions in the Repeat block). If the file supplying data to the instructions in the Repeat still has more data for this repeat block after the maximum loop count is reached then it is an error condition. It should also be noted the addr variable is incremented in the ISC_ADDRESS_SHIFT operation in the Repeat block. The increment occurs before the value is shifted into the device. This means the first time the Repeat block is completed the data shifted in is 0001 (hex). This incremented value is stored in addr and incremented again in the next traversal of the loop.

After the repeat block completes successfully, the Terminate block is executed. In the example above, the terminate block ISC_DATA_SHIFT instruction has a data capture field specified. The exact specification is as follows:

"(ISC_DATA_SHIFT 200:0 wait TCK 100 198:0*0,2:2*3);"

It says the ISC_DATA_SHIFT instruction bit pattern is to be loaded then without traversing Run Test/Idle, 200 bits of 0 are to be loaded in Shift DR. After that has been completed, the state machine is guided to the Run Test/Idle state and 200 TCK pulses are delivered to the device. Then the state machine is directed to the Shift DR state and 200 don't care bits are shifted in. The data output on TDO is compared against the expected result specified. The 200 bits are broken into quantities of 2 and 198 bits. The first two bits out are compared against the value 2 (hex; 10 binary) as pointed out by the value * mask syntax. The digits to the right of the * show the mask. A binary 1 in any mask position signals a significant bit that should be compared. A binary 0 signals the bit value is "don't care". Therefore, the final 198 bits shifted out are not significant as the mask is all 0's.

Using this syntax provides a powerful method for describing configuration memory access operations including all manner or erase, program and read functions.

This syntax also allows devices to be broken up into sectors, each of which is addressed individually. In this manner, a collection of flows can be assembled to describe how to access a device comprised of multiple non-homogeneous sectors with different ISC algorithms.

Another capability within the description is identifying data that contributes to a configuration data cyclic redundancy check (CRC). The CRC can be used to identify the programmed contents of the device. The CRC calculation is associated with reading data from the device. The CRC tags in the description syntax are associated with read back data.

Another important consideration in developing flows to describe configuration algorithms is configuration data size reduction. Keeping the configuration data size small has advantages in embedded systems reducing the run-time data storage needed. The flow syntax includes some arithmetic (add, subtract) and logical (and, or) operations. These can be used to calculate address or sector data reducing the need to store it in the data file.

1.1.9 ISC_Procedure

The ISC_Procedure is used to describe a complex in-system operation. An ISC_Procedure is built by assembling discrete ISC_Flow elements. The ISC_Flow element is just a building block to simplify specifying ISC_Procedures. So the ISC_Procedure is a list of ISC_Flow descriptor elements that are carried out unconditionally and in order of their listing. In addition, ISC_Flow descriptor elements may contain a reference to a data name elements that must match identical data name elements in the ISC data file.

To perform a procedure, the flows in that procedure are performed in order from first to last. A flow is performed unconditionally if it does not take data input. If a flow needs data input, the associated ISC data file is scanned for a matching data name element. If no data with a matching data name is found, an error occurs.

There are sets of predefined procedure names that have meanings as listed below. As a rule, all predefined procedure names begin with "**proc_**".

proc_read: Read the device's memory arrays and output the values, for example, to a file.
proc_verify: Verify the device's memory arrays against user-supplied data typically from the ISC data file.
proc_program: Program the device's memory arrays with user-supplied data typically from the ISC data file.
proc_erase: Bulk-erase the device.
proc_blank_check: Compare the device against its blank state.
proc_enable: Enter ISC mode. The minimal content of this procedure is the ISC_ENABLE instruction.
proc_disable: Exit ISC mode. The minimal content of this procedure is the ISC_DISABLE instruction.
proc_preload: Preload the boundary-scan register using the PRELOAD instruction. It is used for devices with CLAMP-like IO pin behavior during programming. This procedure may be carried out before **proc_enable**. By referencing the standard data name "preload", an interface can determine it is to get state information for the boundary-scan register that is application dependent.
proc_error_exit: This mandatory procedure may be performed by software at run-time when an action is to be canceled (for example, because the user pressed a 'break' key) or some error condition is sensed. The minimal content of this procedure is the ISC_DISABLE

instruction, but ISC_DISCHARGE or other instructions may be placed here. This procedure is used to terminate programming without the ISC_Done bit being set.
proc_program_done: An action calls this mandatory procedure to set the ISC_Done bit.

1.1.10 ISC_Action

In the same way that ISC_Procedures are collections of ISC_Flows, ISC_Actions are collections of ISC_Procedures. The ISC_Procedures that comprise the ISC_Action are carried out in the order of their listing from first to last. The ISC_Procedures that make up each ISC_Action, however, can sometimes be optionally enabled or disabled. Each ISC_Action can be labeled as optional or recommended (or with no designator at all). An optional ISC_Procedure is one that is not executed unless directed to by the end-user. A recommended ISC_Procedure is just the opposite, it is executed unless directed not to by the end-user.

As with ISC_Procedures, there are a set of predefined action names that have meanings as listed below.

read: Read a memory region of a device. The optional <data name> may specify data arrays, IDCODE, USERCODE, and security bits. This can also be used to produce a checksum of the read data using the CRC tags found in the flows.
verify: Verify a memory region of a device's memory arrays, IDCODE, USERCODE, or security bits against user-supplied data.
program: Program a memory region of the device. This action does all the steps needed to install data in a device, whether it is previously programmed or not. Typically, the device is bulk-erased, blank-checked, programmed, and verified.
erase: Bulk-erase the device or erase a region of the device.
blank_check: Compare the device against its blank state.

Besides predefined <action name> elements, the following is allowed.

<identifier>: Perform some operation on the device.

Note the name of operation cannot be localized, and will not necessarily be handled in a standard way on user interfaces of applications that interpret IEEE STD 1532 BSDL files. User interfaces are not in fact required to

display these operations at all. Identifiers should only be used for device-specific functions.

1.1.11 The ISC_Illegal_Exit Attribute

It is preferred that devices that adhere to IEEE STD 1532 allow transitions from ISC instructions to non-ISC instructions without conditions or unusual side effects. This allows interleaving of test and program operations that can reduce overall manufacturing time through test and configuration. This is particularly dramatic when test instructions are used to configure non-IEEE STD 1532 compliant flash memory devices using their surrounding devices' boundary-scan registers.

However, there are situations where practical limits in the design of a programmable logic device prevent the return from a non-ISC instruction to the middle of a programming sequence. The optional ISC_Illegal_Exit attribute describes any instruction that, as a side effect of its execution, clears the ISC_Enabled signal. Because this behavior can be instruction dependent, the ISC_Illegal_Exit attribute allows for specifying only those instructions.

1.1.12 The ISC_Design_Warning Attribute

The optional ISC_Design_Warning attribute may be used, in a manner identical with the Design_Warning attribute of BSDL, to alert users to special circumstances that may exist in the ISC implementation of a given ISC device.

1.2 IEEE STD 1532 BSDL File Example

It is instructive to examine an IEEE STD 1532 BSDL file to better understand the structure and use of such information. Because of this consider the following example. Great portions of this example are identical with IEEE STD 1149.1 BSDL. Namely, the first declaration of the entity is identical. In this case out entity is “FAKE_1532_DEVICE”

```
Entity FAKE_1532_DEVICE is
```

As with an IEEE STD 1149.1 BSDL the device IO ports and package pin mappings are defined. In this case, we are dealing with a PC44 package.

```
Generic (PHYSICAL_PIN_MAP : string := “pc44” );
```

There is a broad collection of pins of all types; power, ground, bidirectionals, inputs, outputs and of course the 4 TAP pins.

```
port ( TDI: in bit; TMS: in bit; Gnd_2: linkage bit; TCK: in bit;
Vcc_1: linkage bit; OUT1: out bit; OUT2: out bit; BIDIR1: inout bit;
BIDIR2: inout bit; BIDIR3: inout bit; BIDIR4: inout bit;
OUT3: out bit; IN1: in bit; Vcc_2: linkage bit; Vcc_1: linkage bit;
Gnd_3: linkage bit;OUT4: out bit ; OUT5: out bit; OUT6: out bit;
Vcc_3: linkage bit; OUT7: out bit; Gnd_4: linkage bit; OUT8: out bit;
TDO: out bit; Vpp: linkage bit; Vcc_4: linkage bit;
Vcc_2: linkage bit; OUT9: out bit; Gnd_1: linkage bit; OUT10: out
bit;
CLK: in bit);
```

Now comes the first hint that this is an IEEE STD 1532 BSDL file. The inclusion of the IEEE_1532_2002 definitions package signals the BSDL may include some attributes defined by that standard.

```
use STD_1149_1_2001.all;
use STD_1532_2002.all;
```

The conformance attribute is specific to IEEE STD 1149.1. Since there is only one version of IEEE STD 1532 there is no specification of the conformance.

```
attribute COMPONENT_CONFORMANCE of
FAKE_1532_DEVICE : entity is
"STD_1149_1_2001"; -- could also be STD_1149_1_1993
```

The pin map shows how the device ports map to its pins. This is a standard IEEE STD 1149.1 BSDL requirement.

```
attribute PIN_MAP of FAKE_1532_DEVICE : entity is
PHYSICAL_PIN_MAP;
constant pc44: PIN_MAP_STRING:=
"TDI:3, TMS:5, Gnd_2:6, TCK:7, Vcc_1:8, OUT1:9, OUT2:10," &
"BIDIR1:13, BIDIR2:44, BIDIR3:1, BIDIR4:20, OUT3:14, IN1:15,"
&
"Vcc_2:16, Vcc_1:17, Gnd_3:18, OUT4:19, OUT5:21, OUT6:25," &
"Vcc_3:26, OUT7:27, Gnd_4:28, OUT8:29, TDO:31, Vpp:35, " &
"Vcc_4:36, Vcc_2:38, OUT9:40, Gnd_1:41, OUT10:42, CLK:43";
```

The TAP signal definition section is also a basic IEEE STD 1149.1 requirement. It points out the maximum frequency of the test clock signal and the <> of the other three TAP signals.

```
attribute TAP_SCAN_IN of TDI : signal is true; attribute
TAP_SCAN_OUT of TDO : signal is true; attribute
TAP_SCAN_MODE of TMS : signal is true; attribute
TAP_SCAN_CLOCK of TCK : signal is (1.00e+07, BOTH);
```

The instruction definition section is identical with that of the IEEE STD 1149.1 BSDL. The instruction register length is defined. Then the list of available instruction bit patterns is defined. The instruction bit pattern patterns are assigned according to the designer's implementation. Grouping the instructions in the file is for clarity only.

```
attribute INSTRUCTION_LENGTH of FAKE_1532_DEVICE :
entity is 8;
attribute INSTRUCTION_BIT PATTERN of FAKE_1532_DEVICE :
entity is
"BYPASS ( 11111111)," &
"SAMPLE ( 00000001)," &
"PRELOAD( 00000001)," &
"EXTEST ( 00000000)," &
"IDCODE ( 11111110)," &
"USERCODE ( 11111101)," &
"HIGHZ ( 11111100)," &
"CLAMP ( 11111010)," &
```

ISC Instructions

```
"ISC_NOOP ( 10011111)," &
"ISC_ENABLE ( 11101000)," &
"ISC_PROGRAM ( 11101010)," &
"ISC_PROGRAM_DONE ( 11101110)," &
"ISC_ADDRESS_SHIFT ( 11101011)," &
"ISC_READ ( 11101111)," &
"ISC_ERASE ( 11101100)," &
"ISC_DATA_SHIFT ( 11101101)," &
"ISC_DISABLE ( 11110000)," &
```

Proprietary ISC Instructions

"VENDOR_BLANK_CHECK (11100101)," &

Private Instructions

"PRIVATE1 (11110001)," &
"PRIVATE2 (11100011)";

The instruction capture attribute points out the value of the bits shifted out of the device as an instruction is shifted in. The two rightmost bits' values are mandated by IEEE STD 1149.1. The next bit to the left is mandated by IEEE STD 1532 to signal the state of the internal ISC_DONE signal. Because this bit's value will change according to the device's state, it is specified as a don't care in the BSDL file. The designer is free to define extra bits that have variable values, as don't cares as well.

attribute INSTRUCTION_CAPTURE of FAKE_1532_DEVICE :
entity is "00XXXX01";

The instruction private attribute defines those instructions that are for private use, typically by the device manufacturer. This is part of the IEEE STD 1149.1 BSDL file.

attribute INSTRUCTION_PRIVATE of FAKE_1532_DEVICE :
entity is "PRIVATE1, PRIVATE2";

Since the IDCODE instruction is mandatory in IEEE STD 1532, the IDCODE register attribute must also be defined. This represents the data value that will be shifted out when the IDCODE instruction is active. This is no different from that of IEEE STD 1149.1.

attribute IDCODE_REGISTER of FAKE_1532_DEVICE: entity is
"0001" & -- version
"0101001000110100" & -- part number
"01011001001" & -- manufacturer's id
"1"; -- required by standard

Like the IDCODE, the USERCODE is mandatory in IEEE STD 1532. This means the USERCODE register attribute must be defined. This represents the data value that will be shifted out when the USERCODE instruction is active. This is no different from that of IEEE STD 1149.1.

```
attribute USERCODE_REGISTER of FAKE_1532_DEVICE : entity
is "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
```

The register access attribute defined the data register that is associated with each instruction. This is identical with that attribute of IEEE STD 1149.1.

```
attribute REGISTER_ACCESS of FAKE_1532_DEVICE : entity is
"DEVICE_ID (IDCODE, USERCODE)," &
"ISC_DEFAULT[1] ( ISC_DISABLE, ISC_NOOP, ISC_ERASE, " &
" ISC_PROGRAM, ISC_PROGRAM_DONE )," &
"ISC_CONFIG[6] ( ISC_ENABLE )," &
"ISC_PDATA[2048] ( ISC_READ, ISC_DATA_SHIFT ),"&
"VENDOR_BLANK[128] ( VENDOR_BLANK_CHECK )," &
"ISC_ADDRESS[16] ( ISC_ADDRESS_SHIFT )";
```

The boundary-scan register definition is defined as in IEEE STD 1149.1. The length and composition of the boundary-scan register is used to facilitate generation of vectors for interconnect test.

```
attribute BOUNDARY_LENGTH of FAKE_1532_DEVICE : entity
is 34;
attribute BOUNDARY_REGISTER of FAKE_1532_DEVICE : entity
is
" 0 (BC_1, CLK, input, X)," &
" 1 (BC_1, *, controlr, 0)," &
" 2 (BC_1, OUT10, output3, X, 1, 0, Z)," &
" 3 (BC_1, *, controlr, 0)," &
" 4 (BC_1, OUT9, output3, X, 3, 0, Z)," &
" 5 (BC_1, *, controlr, 0)," &
" 6 (BC_1, OUT8, output3, X, 5, 0, Z)," &
" 7 (BC_1, *, controlr, 0)," & 1
" 8 (BC_1, OUT7, output3, X, 7, 0, Z)," &
" 9 (BC_1, *, controlr, 0)," &
" 10 (BC_1, OUT6, output3, X, 9, 0, Z)," &
" 11 (BC_1, *, controlr, 0)," &
" 12 (BC_1, OUT5, output3, X, 1, 0, Z)," &
" 13 (BC_1, *, controlr, 0)," &
" 14 (BC_1, OUT4, output3, X, 13, 0, Z)," &
" 15 (BC_1, IN1, input, X)," &
" 16 (BC_1, *, controlr, 0)," &
```

```
" 17 (BC_1, OUT3, output3, X, 16, 0, Z)," &
" 18 (BC_1, *, controlr, 0)," &
" 19 (BC_1, BIDIR1, output3, X, 18, 0, Z)," &
" 20 (BC_1, BIDIR1, input, X)," &
" 21 (BC_1, *, controlr, 0)," &
" 22 (BC_1, OUT2, output3, X, 21, 0, Z)," &
" 23 (BC_1, *, controlr, 0)," &
" 24 (BC_1, OUT1, output3, X, 23, 0, Z)," &
" 25 (BC_1, *, controlr, 0)," &
" 26 (BC_1, BIDIR2, output3, X, 25, 0, Z)," &
" 27 (BC_1, BIDIR2, input, X)," &
" 28 (BC_1, *, controlr, 0)," &
" 29 (BC_1, BIDIR3, output3, X, 28, 0, Z)," &
" 30 (BC_1, BIDIR3, input, X)," &
" 31 (BC_1, *, controlr, 0)," &
" 32 (BC_1, BIDIR4, output3, X, 31, 0, Z)," &
" 33 (BC_1, BIDIR4, input, X)";
```

The section of the BSDL relating specifically to ISC begins after the boundary-scan register definition. These attributes are used by ISC applications to operate the device properly and prepare vectors for configuration of the device in-system.

The ISC_PIN_BEHAVIOR attribute shows how the device pins behave while the device is in ISC mode. Device pins can have two possible behaviors. First, they can float and act as if a HIGHZ instruction instruction were loaded. Second, they can have their behaviors defined by the boundary-scan register contents acting as if a CLAMP instruction were loaded.

```
attribute ISC_PIN_BEHAVIOR of FAKE_1532_DEVICE : entity is
"CLAMP" ; -- clamp behavior
```

The ISC_STATUS attribute points out whether this device includes operation status reporting as described by the standard. Devices that do support this mechanism, when used with compliant software, will have their status monitored automatically.

```
attribute ISC_STATUS of FAKE_1532_DEV ICE : entity is "NOT
IMPLEMENTED" ;
```

The ISC_BLANK_USERCODE attribute shows the value stored as the USERCODE value when it is unprogrammed. This is useful to help tools identify whether the USERCODE has yet been set.

```
attribute ISC_BLANK_USERCODE of FAKE_1532_DEVICE :
entity is "11111111111111111111111111111111";
```

The ISC_FLOW attribute defines the building blocks of the device's configuration access algorithms.

```
attribute ISC_FLOW of FAKE_1532_DEVICE : entity is
```

The names and order of the flows are arbitrary and defined by the designer. In this first flow, flow_program, the "array" tag points out that this flow uses data in the ISC file labeled with the "array" indicator. The flow_program defines how to configure the device pattern memory. Each flow descriptor typically defines an atomic configuration function like erase, program or verify.

```
"flow_program(array) "
"initialize " &
"(ISC_DATA_SHIFT 2048:? wait TCK 1)" &
"(ISC_ADDRESS_SHIFT 16:0000 wait TCK 1)" &
"(ISC_PROGRAM wait 14.0e-3)" &
"Repeat 5 " &
"(ISC_DATA_SHIFT 2048:? wait TCK 1)" &
"(ISC_PROGRAM wait 14.0e-3)" &
```

In this second flow, flow_verify, the same "array" tag is used as in the flow_program. This indicates the device can reuse programming data for verification without modification. IEEE STD 1532 is prejudiced towards devices with this design. Devices that need a different arrangement of data for verify as for program are allowed but would require larger ISC files. In addition data readback from the device cannot be used directly to reprogram another device except using a vendor provided data reformatting tool. This flow uses data in the ISC file labeled with the "array" indicator. The flow_verify defines how to read back the device pattern memory and compare it against the expected configuration data.

Read back verify using auto-incremented address

```
"flow_verify(array) " &
"initialize " &
```

```
"(ISC_ADDRESS_SHIFT 16:$addr=0 wait TCK 1)" &
"(ISC_READ wait 50.0e-6 2048:$data?:CRC)" &
"Repeat 5 " &
"(ISC_READ wait 50.0e-6 2048:$data?:CRC)," &
```

In this third flow, flow_read, the same "array" tag is used as in the flow_program. This flow again uses data in the ISC file labeled with the "array" indicator. In any event, an examination of the flow will reveal that no data is read from the ISC data file for this flow. This is obvious owing to the absence of the "?" in the flow. The array tag is present only for completeness. The flow_read defines how to read back the device pattern memory and dump it to an output indicated by "!". The CRC stag specifies which bits contribute to the device CRC calculation. The CRC calculation formula is fully detailed in the IEEE STD 1532 document.

Read back using auto-incremented address

```
"flow_read(array) " &
"initialize " &
"(ISC_ADDRESS_SHIFT 16:$addr=0 wait TCK 1)" &
"(ISC_READ wait 50.0e-6 2048:!:CRC)" &
"Repeat 5 " &
"(ISC_READ wait 50.0e-6 2048:!:CRC)," &
```

The fourth and fifth flows define how to enter (flow_enable) and exit (flow_disable) ISC mode.

```
"flow_enable " &
"initialize " &
"(ISC_ENABLE 6:4 wait TCK 1)," &

"flow_disable " &
"initialize " &
"(ISC_DISABLE wait 110.0e-3)" &
"(ISC_NOOP wait TCK 1)," &
```

The sixth flow defines how to read and confirm the IDCODE value. The name of this flow is also flow_verify but it indicates that idcode is the data tag. This is a simple example of a secondary method for differentiating between different verify flows. Another application is the situation in which the device program memory is segmented. In that case, data tags can be used to differentiate between different memory segments. It should also be noted

the idcode tag could also be used to point out data to read out of the ISC file to perform the named operation.

Read IDCODE value, mask out version bits

```
"flow_verify(idcode) " &
"initialize " &
"(IDCODE wait TCK 1 32:15434593*0FFFFFFF),"&
```

The seventh flow defines how to erase the device program memory.

```
"flow_erase " &
"initialize " &
"(ISC_ADDRESS_SHIFT 16:0001 wait TCK 1)" &
"(ISC_ERASE wait 100.0e-3)," &
```

The eighth flow defines how to perform a blank check operation. The blank check operation tests if the device program memory is erased.

Test if the device is blank

```
"flow_blank_check " &
"initialize " &
"(VENDOR_BLANK_CHECK wait 50e-3 " &
" 128:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)"&
```

The ninth flow defines how to read and confirm the USERCODE value. The name of this flow is also flow_verify but it indicates that usercode is the data tag. This is a similar use of the data tag as with the idcode flow. It should also be noted the usercode tag could also be used to point out data to read out of the ISC file to perform the named operation.

Compare USERCODE value against blank value

```
"flow_verify(usercode) " &
"initialize " &
"(USERCODE wait TCK 1 32:FFFFFFFF)," &
```

The tenth flow is similar to the ninth excepting the USERCODE value is read out to output indicated by "!".

Compare USERCODE against value read from INPUT

```
"flow_read(usercode) " &
"initialize " &
"(USERCODE wait TCK 1 32:!)," &
```

The eleventh flow says how to program the ISC_DONE signal. This flow programs the control bit (or bits) that enable the external device IOs after programming has completed successfully. This is a mandatory flow and must appear in every compliant device's BSDL file.

```
"flow_program_done " &
"initialize " &
"(ISC_PROGRAM_DONE wait 14.0e-3)," &
```

The twelfth flow is the mandatory flow describing how to exit when an error condition is encountered during execution of any other flow. An error condition is signaled when a data mismatch is detected in any capture field of any flow or when a failure status condition is detected. When an error is detected, the shift is completed and then flow execution is immediately transferred to the flow_error_exit flow. This flow is typically used to set the internal device registers in a benign state and make available specific information about the nature of the failure more readily available.

On any error, erase the device

```
"flow_error_exit " &
"initialize " &
"(ISC_ADDRESS_SHIFT 16:0001 wait TCK 1)" &
"(ISC_ERASE wait 100.0e-3)," &
"(ISC_DISABLE wait 110.0e-3)" &
"(ISC_NOOP wait TCK 1)";
```

Recall that each procedure is made of one or more flows so once the flows are in place the procedures can be built. The procedures identify the middle level building block of device access functionality. The proccdures are used to collect flows and sometimes simplify them. For instance, the "array" data tag of the flow_program, flow_verify and flow_read is removed in its procedure description.

```
attribute ISC_PROCEDURE of FAKE_1532_DEVICE : entity is
"proc_verify(idcode) = (flow_verify(idcode))," &
"proc_enable = (flow_enable)," &
```

```
"proc_disable = (flow_disable)," &
"proc_erase = (flow_erase)," &
"proc_blank_check = (flow_blank_check)," &
"proc_program = (flow_program(array))," &
"proc_verify = (flow_verify(array))," &
"proc_verify(usercode) = (flow_verify(usercode))," &
"proc_read = (flow_read(array))," &
"proc_read(usercode) = (flow_read(usercode))," &
"proc_program_done = (flow_program_done)," &
"proc_error_exit = (flow_error_exit)";
```

Now, with the procedures in place, they are assembled into actions that are the user level macro operations. At the action level, procedures are also tagged with their user options (recommended: that is, default on and optional: that is, default off). Procedures are performed sequentially according to their specification in the action. The user options can be changed by the contents of the ISC data file by using the override records.

```
attribute ISC_ACTION of FAKE_1532_DEVICE : entity is
"erase = (proc_verify(idcode) recommended," &
" proc_enable," &
" proc_erase," &
" proc_blank_check optional, " &
" proc_disable)," &
"program = (proc_verify(idcode) recommended," &
" proc_enable," &
" proc_erase," &
" proc_blank_check proprietary optional, " &
" proc_program," &
" proc_enable," &
" proc_verify optional," &
" proc_disable)," &
"verify = (proc_verify(idcode) recommended," &
" proc_enable," &
" proc_verify," &
" proc_disable)," &
"read = (proc_verify(idcode) recommended," &
" proc_enable," &
" proc_read," &
" proc_disable)";
end FAKE_1532_DEVICE;
```

1.3 Using the IEEE STD 1532 BSDL File

When approaching an IEEE STD 1532-based electronic system, the designer must collect the constituent device 1532 BSDL files and their associated ISC programming data files. A sample chain of IEEE STD 1532 devices is included as Figure 7-3. An application that accesses IEEE STD 1532 devices reads the device BSDL files and stitches them together in a manner to represent the arrangement of devices in their TAP interconnect order. This then is used to represent the device algorithm database. This can be used to set a configuration strategy automatically in two ways. First, the application can use the BSDL and ISC data to configure the device directly. Second, the application can be used to generate the configuration algorithm and data in some intermediate format that can be used and applied to the system some point later in time.

The typical scenario for use of IEEE STD 1532 compliant programmable devices is as follows. First, since IEEE STD 1532 compliant devices are also by definition IEEE STD 1149.1 compliant a single serial chain of IEEE STD 1532 devices may also include IEEE STD 1149.1 devices. This will allow for ease of integration of interconnect test and device configuration. Another alternative would be to separate programmable devices and mask programmed devices into separate chains. This might simplify and streamline device configuration by reducing the number of bypassed devices and the complexity of the TMS and TCK distribution. Unfortunately, it complicates interconnect testing by forcing the synchronization of multiple boundary-scan chains and the development of two independently applied but otherwise dependent sets of test vectors.

Once the chain architecture is in place, for each IEEE STD 1149.1 device a BSDL file is needed. If bypassing the device, the instruction register length is all that is needed since the bypass instruction is standardized to be all 1's. As well, the IEEE STD 1532 BSDL and ISC data files are needed for each IEEE STD 1532 device.

To set up device configuration, the application software accepts all the BSDL and ISC data files and then, according to the user-specified actions and options, produces a vector stream that carries out the device operations. A sophisticated application will be able to optimize system configuration times by performing these operations in concurrent mode. What this means is the application is able to coordinate device burn times in a manner to allow many devices to program (or erase or read) locations simultaneously.

If this optimization is done intelligently, then many devices can be configured in the same time it takes for a single device.

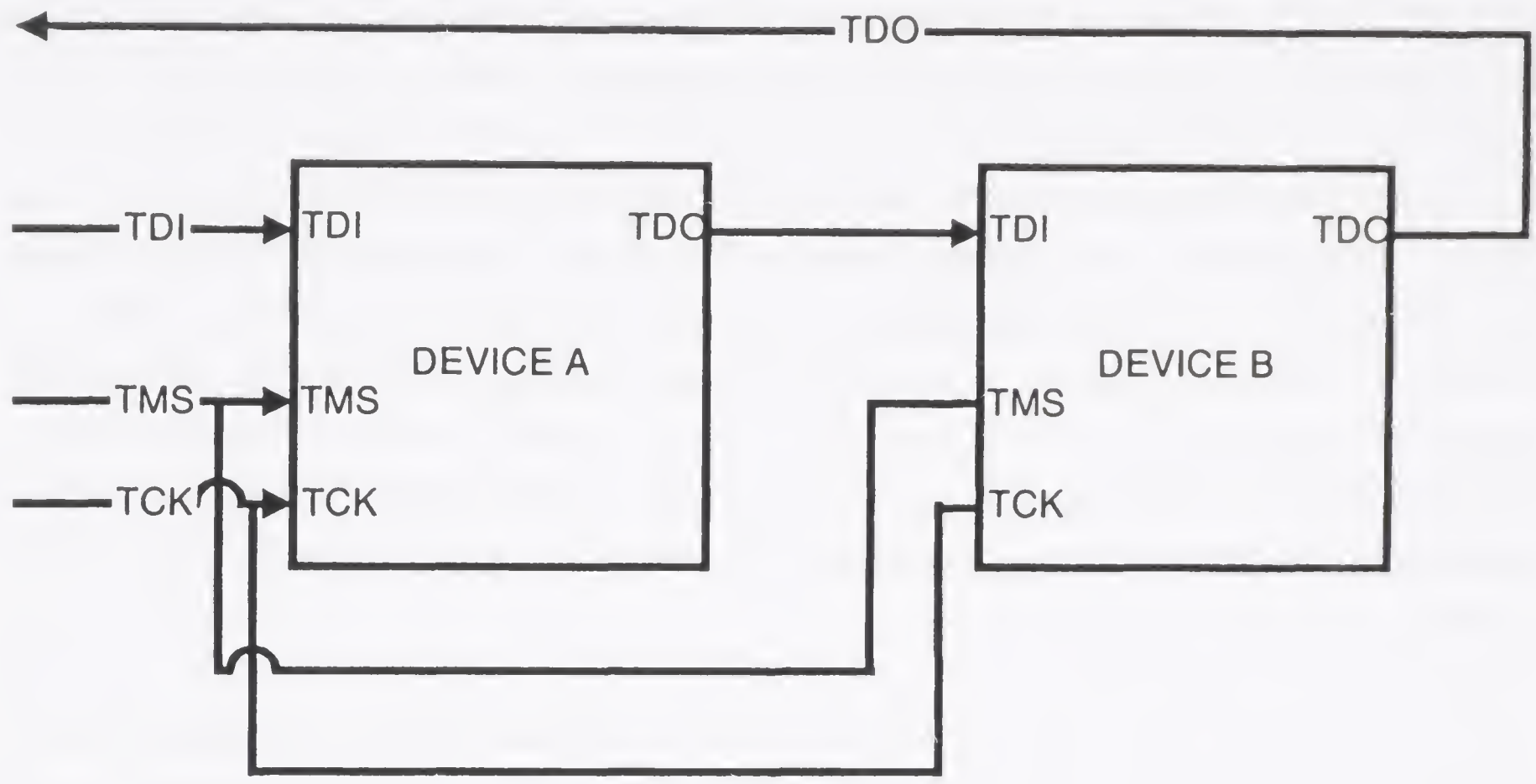

Figure 7-3. A Sample Multiple Device Chain

To better understand this, consider the following examples. In the first scenario, we look at the most likely and most simple situation, that of two identical devices:

Device 1:
Repeat 50
ISC_PROGRAM 32:? Wait 10 msec
Device 2:
Repeat 50
ISC_PROGRAM 32:? Wait 10 msec

A simple method for developing a concurrent flow would look like this:

Repeat 50
ISC_PROGRAM, ISC_PROGRAM 32:?, 32? Wait 10 msec

Neglecting the contribution of shift times (which might not always be accurate, but more on this later), the total configuration time would be 50 * 10 msec or 500 msec. Compare this against the sequential configuration time of 50 * 10 msec + 50 * 10 msec or 1 second. Concurrency represents a system configuration time savings of roughly 50%. As more identical devices are added the total concurrent configuration time remains 500 msec -

as long as you can shift the bits in fast enough relative to the wait time of 10 msec. In fact, if your TCK is running at 10 MHz then 10 devices will only contribute 320 shifts or 32 microseconds for each ISC_PROGRAM instruction. This can clearly provide significant throughput improvement when applied to a manufacturing line processing thousands or tens of thousands of these devices.

Let's look at a slightly more complicated example. In this scenario, we have two devices with similar program flows but different burn times and sizes.

```
Device 1:
Repeat 50
ISC_PROGRAM 32:? Wait 10 msec
Device 2:
Repeat 25
ISC_PROGRAM 64:? Wait 15 msec
```

A simple method for developing a concurrent flow would look like this:

```
Repeat 25
ISC_PROGRAM, ISC_PROGRAM 32:?, 64? Wait 15 msec
Repeat 25
ISC_NOOP, ISC_PROGRAM 1:1, 32:? Wait 10 msec
```

Neglecting the contribution of shift times, the total configuration time would be 25 * 15 msec + 25 * 10 msec or 625 msec. Compare this against the serial configuration time of 50 * 10 msec + 25 * 15 msec or 875 msec. Concurrency represents a system configuration time savings of around 30%.

A more complicated example has some more interesting possibilities as explained below:

```
Device 1:
Repeat 25
ISC_ADDRESS_SHIFT  36:$addr+1 Wait TCK 1
ISC_DATA_SHIFT 1024:? Wait TCK 1
ISC_PROGRAM                Wait 100 microseconds
Device 2:
Repeat 50
ISC_PROGRAM 256:? Wait 10 microseconds
```

One approach to assign a concurrent flow would be to combine the large wait times together coordinating the ISC_PROGRAM instructions. This would look like this:

```
Repeat 25
ISC_ADDRESS_SHIFT, ISC_NOOP 36:$addr+1, 1:0 wait TCK 1
ISC_DATA_SHIFT, ISC_NOOP 1024:?, 1:0 wait TCK 1
ISC_PROGRAM, ISC_PROGRAM 1:0, 256:? wait 100 microseconds
Repeat 25
ISC_NOOP, ISC_PROGRAM 1:0, 256:? wait 10 microseconds
```

Neglecting the contribution of the shift times, the total configuration time would be 25 * 100 microseconds + 25 * 10 microseconds or 2.75 milliseconds. When compared against the sequential time of 25 * 100 microseconds + 50 * 10 microseconds or 3 milliseconds you see a less dramatic but still measurable savings. Further efficiencies can be squeezed out of concurrency as follows:

```
Repeat 16
ISC_ADDRESS_SHIFT, ISC_PROGRAM 36:$addr+1, 256:? wait 10
microseconds
ISC_DATA_SHIFT, ISC_PROGRAM 1024:?, 256:? wait 10
microseconds
ISC_PROGRAM, ISC_PROGRAM 1:0, 256:? wait 100 microseconds
Repeat 1
ISC_ADDRESS_SHIFT, ISC_PROGRAM 36:$addr+1, 256:? wait 10
microseconds
ISC_DATA_SHIFT, ISC_PROGRAM 1024:?, 256:? wait 10
microseconds
ISC_PROGRAM, ISC_NOOP 1:0, 1:0 wait 100 microseconds
Repeat 8
ISC_ADDRESS_SHIFT, ISC_NOOP  36:$addr+1, 1:0 Wait TCK 1
ISC_DATA_SHIFT, ISC_NOOP 1024:?,1:0 Wait TCK 1
ISC_PROGRAM, ISC_NOOP 1:0,1:0  Wait 100 microseconds
```

In this variation, all the operations of Device 1 are combined in all steps of Device 2. The total time is therefore 17 * (100 microseconds + 10 microseconds + 10 microseconds) + 8 * 100 microseconds or 2.84 milliseconds. This is slightly longer than the previous version. But if TCK is running at less than 100 KHz then each TCK pulse takes 10 microseconds or

longer. By combining the ISC_DATA_SHIFT and ISC_ADDRESS_SHIFT operations with Device 2's ISC_PROGRAM, you increase configuration throughput. Redoing the numbers with wait TCK 1 taking 10 microseconds, you get the sequential time is 3.5 milliseconds. The first approach reduces the total configuration time to 3.25 milliseconds but the second approach reduces the total configuration time to 3 milliseconds.

If the TCK period is 10 microseconds then it is also true that a shift of 1024 bits cannot be safely ignored in calculating the configuration time throughput. A shift of this length will take 10.24 milliseconds. In this situation, the shift time is significantly greater than the wait time. It might be the case that at slow TCK speeds when there is much shifting to do and when the program wait times are small, sequential configuration will be the fastest.

This short example explains some of the parameters that must be evaluated by IEEE STD 1532 applications to discover the ideal collection of devices to carry out concurrent configuration efficiently. It may in fact be the case that for some device groupings, concurrent configuration may be slower than sequential configuration. This will likely be the case when the TCK speeds are slow and the disparity between device wait times is great.

The time saved in these examples seems small. These examples, however, are for small amounts of data and small numbers of devices. In larger groups of larger devices and across large numbers of boards typical of a manufacturing run, the time saving will be large and translate into significant dollar savings.

Having devices that are IEEE STD 1532 compliant enables the use of concurrency. Devices of this sort are guaranteed to be well behaved and not to be damaged if their burn times are exceeded or too many TCK pulses are applied as may well happen, when grouped with other devices in concurrent mode.

Applications that use the IEEE STD 1532 BSDL and ISC files directly as in Figure 7-4, have certain significant advantages over those that create an intermediate file as in Figure 7-5. The key advantage is that they are more easily adaptable to changing scenarios that are typical of the board maintenance process. For instance, during initial board programming you collect all BSDL and ISC files and produce a single concurrent configuration description in some intermediate form. Later, however, if one or two design patterns change then you have some decisions to make. You will have a

series of boards that need to be updated and new boards that need to be fully programmed. Using an application that needs an intermediate file would require generation of two new files. One file that configures just the updated design patterns and one that does a full concurrent configuration of the new boards. This full reconfiguration file must use the new patterns for the changed devices and the original patterns for the unchanged devices. If the application simply uses the BSDL and ISC files directly then no extra file generation and tracking is needed. The source BSDL and ISC files are used directly.

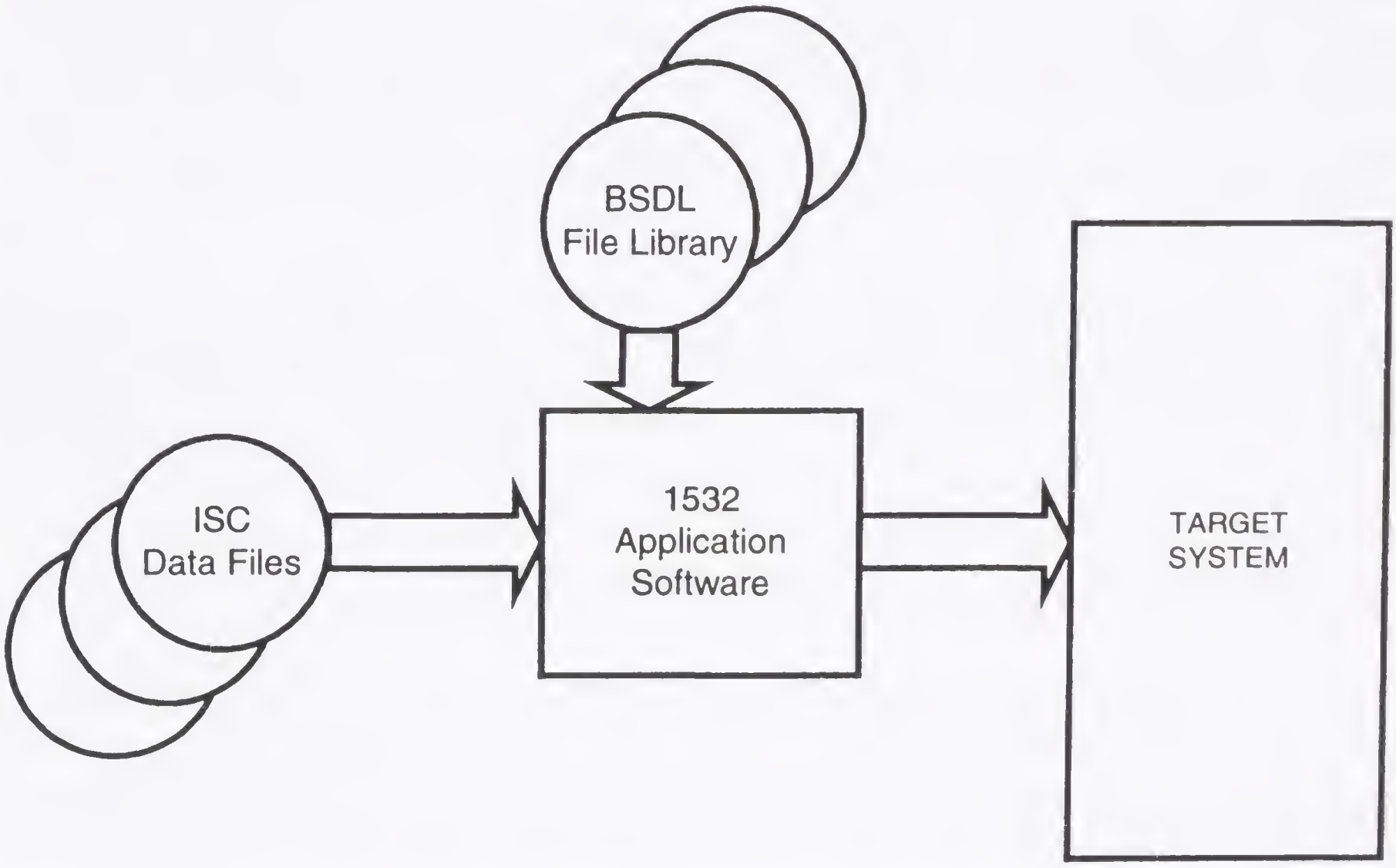

Figure 7-4. User flow when BSDL and ISC files are used directly

An intermediate file has an advantage in a system in which changes are unlikely to occur in that the configuration algorithm is produced once and reused with each application. This saves the processing time associated with recalculating the configuration algorithm.

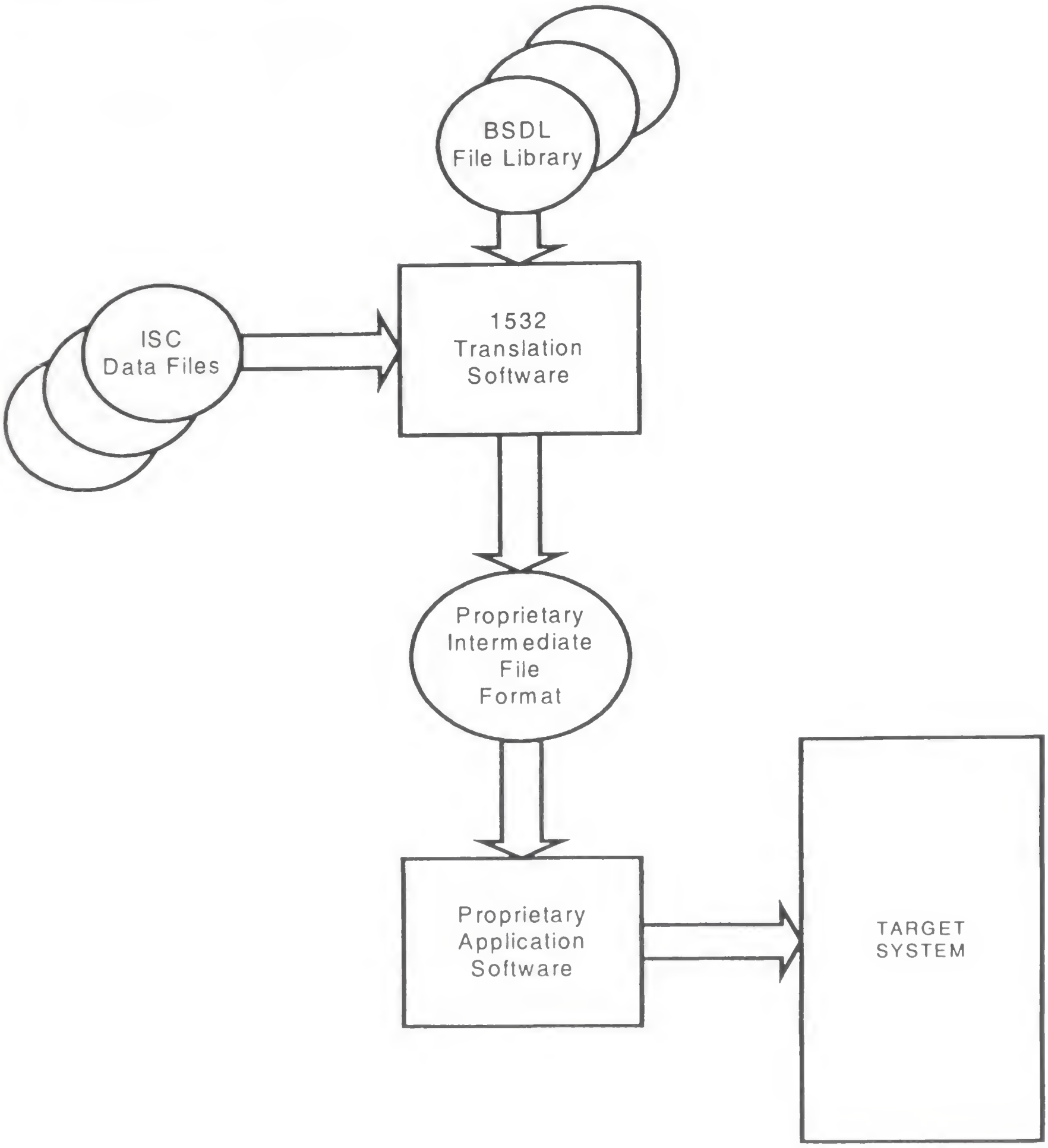

Figure 7-5. User flow when Intermediate files are used

A good application should allow the end user to select the mode of operation that is best for their situation and temperament. For instance, allowing intermediate file use during production runs and direct use of BSDL and ISC files during algorithm or data update.

2. Comparative Evaluation of Approaches

We have examined several different descriptions and mechanisms for the storage and maintenance of configuration data for programmable devices. In summary, they have the following characteristics:

- JEDEC
  - Contains configuration data only in ASCII readable form
  - No algorithmic information
  - Limited data compression capability
  - Often adhered to as a data format only with different vendors interpreting its contents differently
  - Used primarily by device programmer manufacturers
- SVF
  - Integrated data and algorithm in ASCII readable format
  - No control flow in algorithmic description – straight-line execution only
  - Limited data compression capability
  - De facto standard for interchange of boundary-scan data flows.
  - Widely supported and accepted
  - Used primarily by boundary-scan tool developers, ATE manufacturers and embedded systems programmers
- STAPL
  - Integrated data and algorithm in ASCII readable format
  - Basic algorithmic control flow
  - Standardized data compression
  - JEDEC standard but with limited support and acceptance.
  - Used primarily by embedded systems programmers
- Java API for Boundary-Scan
  - Separable data and algorithm using Java programming language
  - All Java-supported control flow mechanisms
  - Standardized but extensible data compression
  - Informal standard with limited support and acceptance
  - Used primarily by embedded systems programmers
- IEEE STD 1532
  - Separate data and algorithm using IEEE STD 1149.1 BSDL extension and new ISC data format
  - Control flow limited to counted loops and loop on condition
  - Limited data compression
  - Widely accepted IEEE standard with significant momentum

- Used primarily by boundary-scan tool developers, ATE manufacturers and embedded systems programmers with device programmer manufacturers looking into support

The systems designer must answer the question: "What will work best for my application and me?" As you might expect the answer is not as straightforward as might be desirable.

What is clear is that JEDEC should be avoided. This file format has been too broadly interpreted to be useful. There is inadequate information in a JEDEC file to complete device programming successfully. You will need significant added information and guidance from the device vendor to use JEDEC files effectively. You should avoid these files at all costs.

SVF files serve as the common interchange format of the in-system configuration community now. Though simple, these files can describe all devices effectively and with relative efficiency. Some devices, typically those based on flash technologies, are not accurately describable using SVF files. Vendors who supply these devices and claim to describe their programming in SVF have proprietary interpretations of SVF or may require you to buy specially sorted devices to guarantee correct configuration.

Although a formal JEDEC standard, STAPL remains closely associated with Altera. Most other vendors still view it with some suspicion. Altera remains the key proponent of the format. They still however produce STAPL or SVF files from their applications to describe the configuration of their devices. In addition, they support IEEE STD 1532.

Most users who have found STAPL useful are embedded system programmers who use it to effect programming of their nonvolatile PLDs. Run time memory remains an issue and depending on the interpreter that you use, you may be limited to sequential device programming. As suggested previously, this might be preferred, when updates are expected. The available interpreters are good although those with a good programming background may be able to make speedier and more efficient implementations. This might be important for memory-sensitive applications. Support for STAPL remains spotty, Altera is you best source for answers on issues related to STAPL.

Java API for Boundary-Scan occupies a smaller niche. It has found its sweet spot in Java's market space – that of embedded or desktop internet-

connected applications. Support for the API is limited; the author of this text is the best source of answers on issues.

IEEE STD 1532 is rapidly gaining acceptance in the marketplace. Support is being quickly rolled out. There is little dissonance among the vendor community about supporting the standard so this bodes well for its future. Because of its broad acceptance and its IEEE standard status, there are many points of contact for support. You should first approach the IEEE STD 1532 working group. The IEEE web site contains contact information for that group. Several implementations of kernels to interpret IEEE STD 1532 BSDL and data files are available. They have been successfully ported to various embedded processor platforms.

The following table summarizes the sweet spots for each solution:

Table 7-1. Configuration Description Language Application Spaces

Solution	ATE	Embedded Systems	Boundary-Scan Tools	Device Programmers
JEDEC	NO	NO	NO	YES
SVF	YES	MAYBE	YES	NO
STAPL	MAYBE	YES	MAYBE	NO
JAVA	NO	YES	NO	NO
IEEE STD 1532	YES	YES	YES	MAYBE

Chapter 8

The IEEE STD 1532 Compliant Device

1. Introduction

We have now seen the software infrastructure that underpins IEEE STD 1532. Now we will describe what an IEEE STD 1532 compliant device looks like to the end user.

2. Operating States

A device compliant with IEEE STD 1149.1 has two distinct operating states. It is either in test mode and the boundary-scan register controls the pin states or it is in mission mode and the device function controls the pin states. This is pictured in Figure 8-1.

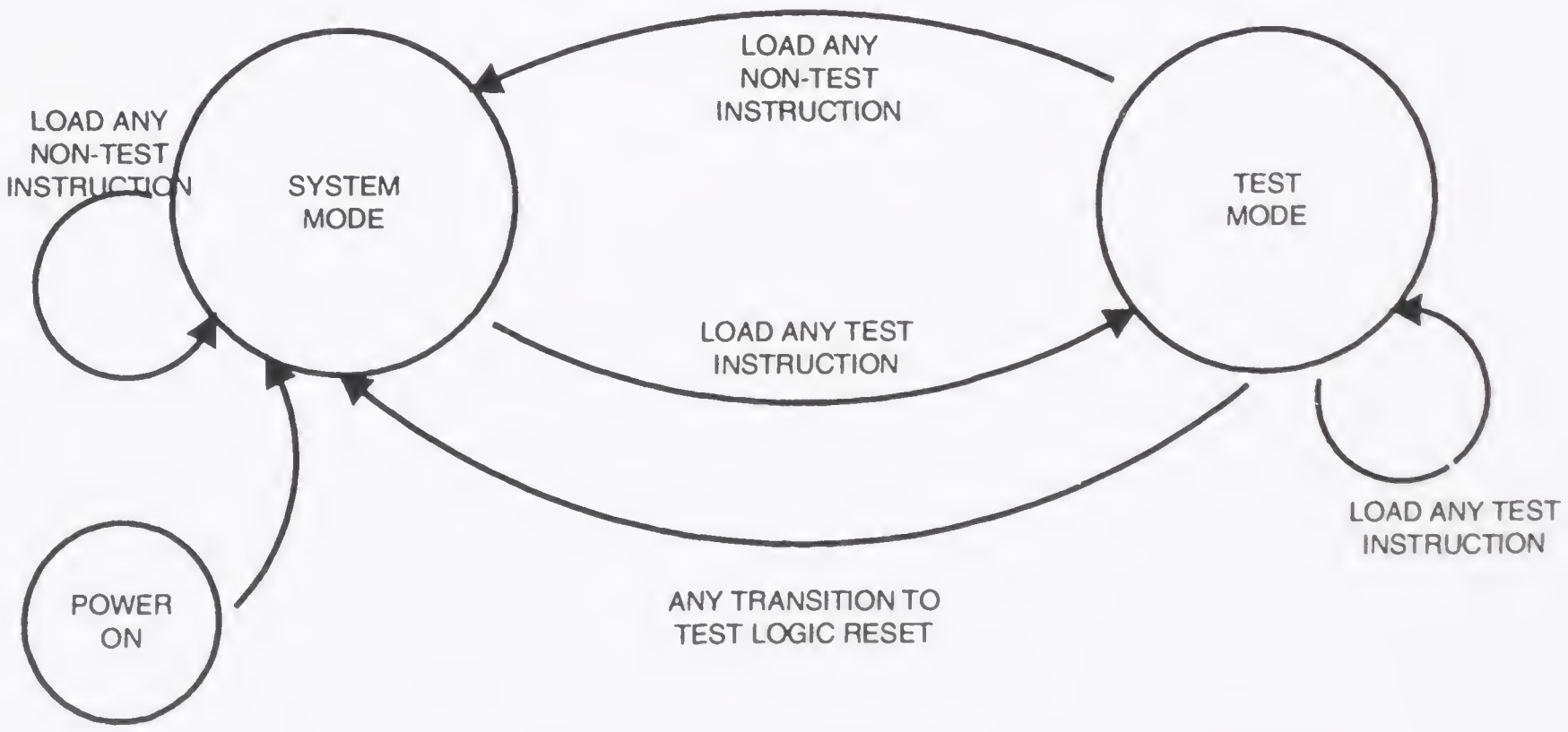

Figure 8-1. Operating Modes for IEEE STD 1149.1 Compliant Devices

A programmable device that is compliant with IEEE STD 1532 (and therefore, by definition also with IEEE STD 1149.1) overlays another two states. It is either being programmed or it is in operation. It turns out that

when a programmable device is used in a system the states need more refinement.

IEEE STD 1532 defines four such states that it refers to as the system modal states:

- **Unprogrammed** – In this state, a device is either blank or incompletely programmed.
- **ISC Accessed** – In this state, the device's configuration memory is being accessed for erasing, programming or reading
- **ISC Complete** – In this state, the configuration operations have been completed but the device is not yet operational. The device remains in this state as long as the ISC_DISABLE instruction is loaded in the device's instruction register. This allows controlled sequencing of devices to the operational state.
- **Operational** – In this state, the device's behavior is fully defined by the programming patterns loaded into the devices configuration memory.

A typical sequence of transitions is as follows. The device powers up and is blank. It is therefore in the **Unprogrammed** modal state. According to the standard, the programmable pins of the device should be floating.

Now the designer wants to program the device. Loading the ISC_ENABLE instruction completes the transition to the ISC Accessed modal state. Once in the ISC Accessed modal state, all operations that access the device's configuration memory can be completed. The ISC_ENABLE instruction has one of two possible behaviors on activation. The device's programmable pins either float or clamp to values determined by the contents of the boundary-scan register. The behavior of the pins is pointed out by the ISC_PIN_BEHAVIOR attribute in the BSDL file.

Device erasure, programming and verification are completed in the ISC accessed modal state. An IEEE STD 1532 compliant device will have a special bit (or group of bits) programmed that signals the device has been configured successfully. This bit is known as the ISC_Done. After programming of the ISC Done bit, the device is ready for operation.

The ISC_DISABLE instruction is loaded to prepare the device for full operation. While the ISC_DISABLE instruction is loaded, the device is in the **ISC Complete** modal state. When the ISC_DISABLE instruction is displaced from the instruction register (by loading a BYPASS or other non-

ISC instruction) the device transitions to the **Operational** modal state. The device now takes on its programmed behavior.

A modal state diagram that more fully explains the transitions is included as Figure 8-2.

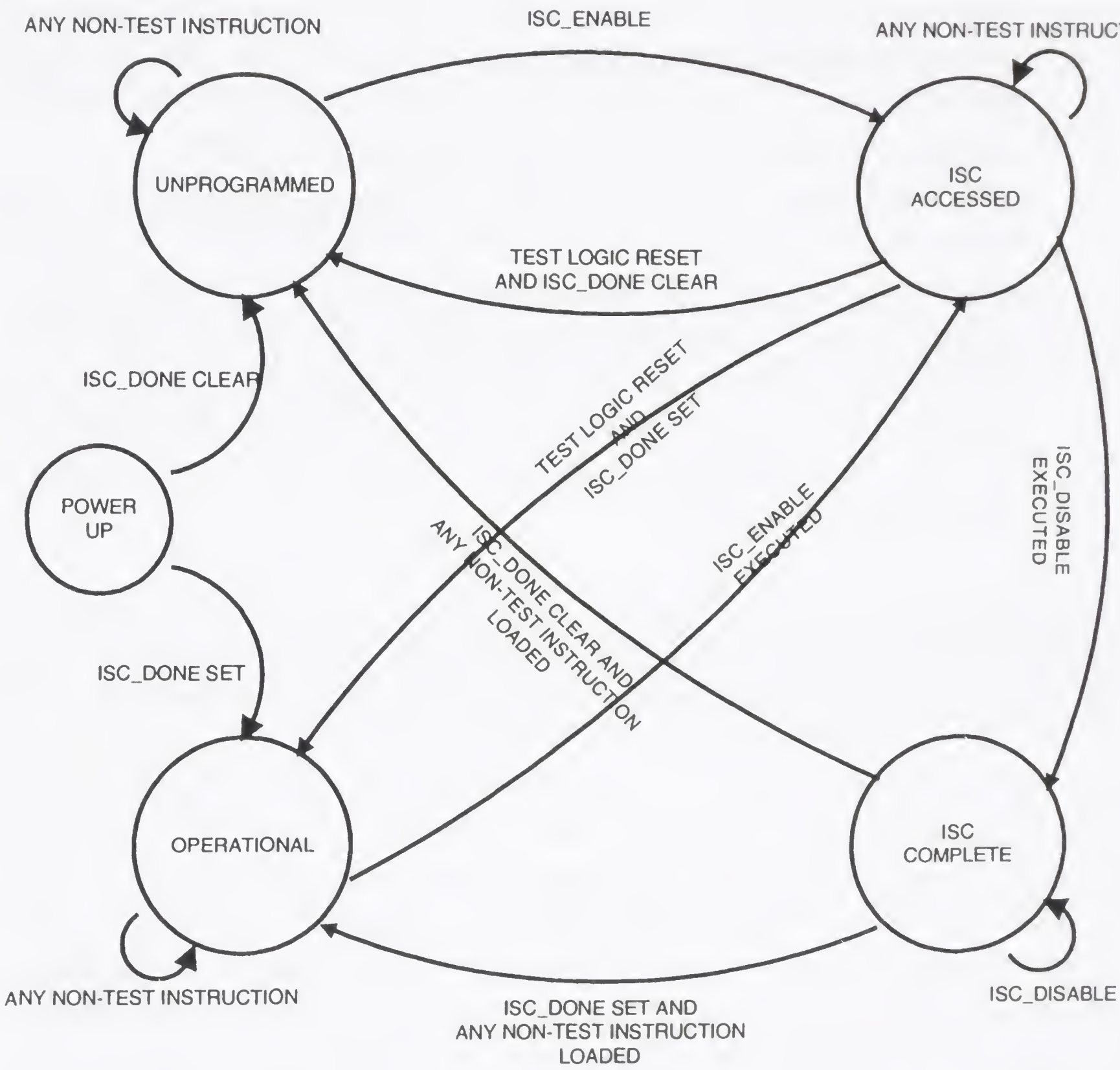

Figure 8-2. IEEE STD 1532 Configuration Modal State Transition Diagram

3. System Pins

IEEE STD 1532 carefully describes a classification of system pins. There are five pin types:

- **Compliance Enable Pins** – these pins are identical with the compliance enable pins of IEEE STD 1149.1. These pins are

used with static deterministic logic state conditions to enable device compliance with IEEE STD 1149.1.

- **Test Access Port Pins** – these are the four pins of the Test Access Port (TCK, TMS, TDI and TDO).
- **Programming Voltage Pins** – These optional special programming voltage pins can be used to provide access for over-voltage programming.
- **Fixed System Pins** – These are pins, earmarked by the device designer, whose IO functions are not determined by the configuration information programmed into the device.
- **In-System Configurable System Pins** – These are pins, whose IO functions are determined by the configuration information programmed into the device.

Of all the pin types described above, only the In-System Configurable System Pins are affected by the ISC_ENABLE instruction and described by the ISC_PIN_BEHAVIOR attribute. These pins also remain three-stated when the device is either erased or incompletely programmed. Incompleteness is determined by the state of the ISC_Done bit. If programmed then the configuration was completed if not the device should look and behave like an erased device.

Fixed system pins are listed in the FIXED_SYSTEM_PIN attribute in the BSDL file. All the other pin types are covered by the rules associated with IEEE STD 1149.1 BSDL.

4. Algorithmic Operation

IEEE STD 1532 compliant devices have some strict rules to which they must adhere in all configuration operations. These rules are key to allowing the devices to work well in a system. They also allow concurrent algorithm application to speed programming throughput.

4.1 Algorithm Steps and State Transitions

The basic algorithm step is as previously described. It is a group of four steps consisting of an instruction load, an input data shift, a wait in Run Test/Idle and an output data shift. A sequence of algorithm steps is performed to complete a configuration operation.

The device cannot require any specific state trajectory be followed between each step. For instance, the device cannot need a wait in Pause DR between the data shift and the wait in Run Test/Idle. Conversely, it should tolerate all valid transitions in performing the algorithm steps. Therefore, it should allow a Shift DR to be interrupted by a visit to Pause DR and a traversal back to Shift DR to complete the step.

The step that includes a wait time in Run Test/Idle is always required. The instruction operations should be carried out in the Run Test/Idle state. Devices should tolerate longer than specific waits without causing any damage. This allows operations to be performed concurrently.

4.2 Algorithm Optimizations

Devices should also be tolerant of step optimizations made by the applications software interpreting the BSDL files. This includes optimizations of the following types:

- **Deleting Redundant Instruction Loads** – In situations in which the same instruction is active for multiple steps, there is no need to reload the instruction with each step. For instance:

  Repeat 25
  (ISC_PROGRAM 25:? Wait 10e-3 25:3*7)

  Execution of this fragment need not repeat the load of ISC_PROGRAM with each of the 25 steps. It can be loaded for the first step and then the data can be loaded with transitions directly to Shift DR. Schematically the flow looks like this:

  Shift IR: ISC_PROGRAM
  Shift DR: Shift in 25 bits read from file.
  Run Test/Idle: Wait 10 msec.
  Shift DR: Shift out 25 bits test first three bits out against 3 hex.
  Skip Run Test/Idle
  Shift DR: Shift in next 25 bits read from file.
  Run Test/Idle: Wait 10 msec.
  Shift DR: Shift out 25 bits test first three bits out against 3 hex.
  Skip Run Test/Idle

Shift DR: Shift in next 25 bits read from file.
etc.

- **Interleaving Data Input and Output Shifts** – In a multiple step operation in which data must be both shifted into and out of the device, the first output data shift can be interleaved with the second input data shift. More generally, the Nth output data shift can be interleaved with the N+1st input data shift. The above example simplifies as follows:

  Shift IR: ISC_PROGRAM
  Shift DR: Shift in 25 bits read from file.
  Run Test/Idle: Wait 10 msec.
  Shift DR: Shift in next 25 bits read from file - Shift out 25 bits test first three bits out against 3 hex.
  Run Test/Idle: Wait 10 msec.
  Shift DR: Shift in next 25 bits read from file - Shift out 25 bits test first three bits out against 3 hex.
  etc.

- **Arbitrary ISC_NOOP insertion** – Often, to get concurrent operation to work, extra ISC_NOOP instructions may need to be inserted in the algorithm flow. Devices should tolerate this without causing the current operation to fail. For instance the above flow should still work with ISC_NOOPs added as shown:

  Shift IR: ISC_PROGRAM
  Shift DR: Shift in 25 bits read from file.
  Run Test/Idle: Wait 10 msec.
  Shift DR: Shift out 25 bits test first three bits out against 3 hex.
  Skip Run Test/Idle
  Shift IR: ISC_NOOP
  Shift DR: Shift in NOOP bits
  Run Test/Idle: Wait arbitrary time
  Shift DR: Shift out NOOP bits
  Skip Run Test/Idle
  Shift IR: ISC_PROGRAM
  Shift DR: Shift in next 25 bits read from file.
  Run Test/Idle: Wait 10 msec.
  Shift DR: Shift out 25 bits test first three bits out against 3 hex.
  Skip Run Test/Idle
  Shift IR: ISC_NOOP

Shift DR: Shift in NOOP bits
Run Test/Idle: Wait arbitrary time
Shift DR: Shift out NOOP bits
Skip Run Test/Idle
Shift IR: ISC_PROGRAM
Shift DR: Shift in next 25 bits read from file.
etc.

Note the ISC_PROGRAM instruction must be reloaded each time after the ISC_NOOP is loaded.

4.3 Proprietary Algorithm Support

Some devices may have algorithms that can be described using IEEE STD 1532 BSDL but the devices lack compliant electrical features making them unsuitable for concurrent operations or algorithmic step optimization. These devices have the **proprietary** keyword associated with the procedures listed in an action or the action itself.

4.4 Nullified Instructions

There is a provision in the standard for nullifying instructions. This occurs when an ISC instruction is loaded when a device is not in the ISC Accessed modal state. Externally the ISC instruction behaves according to the ISC_PIN_BEHAVIOR attribute but internally the operation (say, ISC_PROGRAM) is not completed. In addition, there is no damage to the device or its current programmed contents.

4.5 Interleaving Test and Configuration Instructions

The default behavior of IEEE STD 1532 compliant devices is that transition between ISC and test mode instructions are allowed. This means that you could interleave a test operation with an EXTEST. This might be useful in situations in which EXTEST is used to access the functional pins of adjacent non-IEEE STD 1532 FLASH memory devices for programming.

These sorts of transitions between ISC and test mode may be risky to the ISC device. The attribute ISC_ILLEGAL_EXIT is used to list the test mode instructions that cannot be used during ISC operations.

4.6 Asynchronous Transitions to Test Logic Reset

Asynchronous transitions to Test Logic Reset using the TRST pin during ISC operations are intended as a panic escape route. The device behavior is similar to performing an ISC_DISABLE. Usually ISC_DISABLE has a wait time associated with it. This may not be guaranteed if you assert the TRST pin. In that case, the only fact known is the device will exit the ISC Accessed modal state.

4.7 Device Operation Status Indication

The standard recommends that devices always return some operation status information. The standard describes status as a reflection of the mechanics of the configuration operation. This means that it signals whether the device is in ISC Accessed state, suitable time was spent in Run Test/Idle and valid data was supplied to the operation. However, it does not reflect the success of the configuration operation itself.

According to the standard, any designer developed approach for status collection (including none) is acceptable. However, the standard describes one method that can be handled automatically by IEEE STD 1532 applications.

The specified method requires that all data registers have status bits assigned in them. These bits can be queried by the application when the data register contents are shifted out. The status bits must always be in the same location – regardless of the active instruction. The minimum number of status bits is two. The two status bits must be the first two bits shifted out (the two least significant data register bits). The status code "10" says no error has occurred and the code "01" says some error has occurred. The other bit code patterns ("11", "00") are illegal. By choosing status codes that are the inverse of one another and with a 0 and 1 pattern, electrical issues can be easily detected.

The status bits are detected automatically by application software and if an error occurs, **proc_error_exit** is performed.

An optional status subcode field of arbitrary size can be used for the device to provide added information about the failure signaled. This cannot be dealt with by applications automatically but applications should allow end users to view the subcode contents using a captured data log.

4.8 Device Operation Success Indication

To provide support for devices with non-deterministic configuration algorithms, the device must be able to signal when it needs extra configuration attempts. The operation success indication serves that role. These indicators exist in addition to any operation status indication bits. The operation success indication bits – unlike the status indication bits – do reflect the success of the configuration operation itself.

These bit values need to be clearly tagged using the OST keyword in the BSDL. This points out to the application that these bits signal whether a location has been successfully configured. Consider this flow:

```
(ISC_PROGRAM 32:? Wait 10e-3)
loop min 10 max 100 (
(ISP_NOOP wait 10e-3 1:1:OST, 2:0*0)
)
```

The ISC_NOOP instruction is used to wait for the operation started by the ISC_PROGRAM instruction to complete. The **loop** section of the flow shows the success indication bit is the 3rd bit shifted out. The programming will be complete when that bit is sensed as a logic 1. The loop will continue executing up to 100 times (indicated by **max**). If the OST specified bit is not 1 after 100 iterations it is an error condition. The **min** keyword shows that OST bit need not be tested until the 10th loop iteration.

Interestingly, the standard also allows loops of this sort to be expanded to their maximum specified number of steps. Doing this should not damage the device. Note though that each step must be performed. Because the max is specified to be 100, it is not accurate to say the loop statement:

```
loop min 10 max 100 (
(ISP_NOOP wait 10e-3 1:1:OST, 2:0*0)
)
```

equals:

```
(ISP_NOOP wait 1000e-3 1:1:OST, 2:0*0)
```

Rather it equals:

```
(ISP_NOOP wait 10e-3 1:1:OST, 2:0*0)
```

repeated 100 times.

5. Summary

Adherence to IEEE STD 1532 is a contract. It means that users can expect that designers have provided certain specific device behavior and functionality. It also means that configuration application developers have a good deal of functionality they too can provide.

Chapter 9

DESIGN CONSIDERATIONS FOR IN-SYSTEM CONFIGURABLE SYSTEMS

1. Introduction

In this chapter, we will consider design rules for configurable systems. This includes first figuring out the proper configurable device, then designing the infrastructure for the device including wiring the signals and providing suitable power. After that, we will examine considerations for integrating test and configuration and explore the class of configurable system needed.

2. Device Selection Criteria

When trying to select a device suitable for use in a configurable system there are many design considerations:

1. IEEE STD 1532 Compliance
   a. Fail safety
   b. Support of Concurrency
2. Power consumption during configuration
3. Configuration Speed
4. Endurance
5. Data Retention
6. Security
7. Reliability
8. System Boot Time
9. Initialization
   a. After Power Interruption
10. Configuration Process Validation

Each of these matters will be discussed in sequence.

2.1 IEEE STD 1532 Compliance

You need to decide if the benefits of IEEE STD 1532 compliance are important and valuable to you in the system you are designing. Briefly, the specific benefits are multi-vendor device support including the ability of doing concurrent device configuration.

Devices that fully comply with IEEE STD 1532 have ISC_Done meaning that if power fails during configuration, devices will not power up in an unsafe state. ISC_Done provides a high degree of system protection if configuration fails or a power interruption occurs.

In addition, there is the benefit of a single data interface for all devices using potentially a single application. Separating data and algorithm information provided directly from the device manufacturer ensures a high degree of confidence in the validity of the programming actions and promotes simple update of either data or algorithm or both.

2.1.1 IEEE STD 1532 Compliant vs. IEEE STD 1532 Compatible

IEEE STD 1532 recognizes the capacities defined in the specification are essential for the programming devices from multiple vendors on a single chain. The standard therefore states that "A component *conforming* to this standard shall comply with all rules set herein". This is not to be taken lightly or confused with *being compatible*, a term suggesting that a device may follow the standard but deviates in some ways that may be significant.

With the arrival of IEEE STD 1532, many manufacturers are bringing out devices labeled as conforming to IEEE STD 1532. While usually this is true, there may be significant deviations from the standard in a minority of devices. The terminology accepted for these latter devices is that they are IEEE STD 1532 Compatible (that being a weaker form of compliance).

Device manufacturers typically identify the areas of non-conformance that warrant the compatibility label either in their data sheets or in the BSDL files. If not, an IEEE STD 1532 compliant device has the following characteristics:

1. It **fully** complies with IEEE STD 1149.1
2. Its configurable pins have predefined behaviors before during and after configuration

3. It does its "configuration work" while waiting in the Run Test/Idle TAP controller state
4. It does not need any specific TAP state sequencing to configure correctly
5. It fulfills the DONE functionality ensuring that partially programmed devices will not "wake up" in a partially functional state.
6. It can be ISC accessed concurrently with other IEEE STD 1532 device to improve system configuration throughput

The device BSDL files contain telltale signs of compatibility. The first is the existence of the keyword **proprietary** in any algorithm description. This disallows concurrent operation of these devices. Applications cannot make algorithmic optimization of flows or actions sections marked **proprietary**. This typically occurs when the device has an algorithm that is describable in BSDL but does not conform to the strict requirements of IEEE STD 1532.

Another sign is the absence of a program-done flow, or having a program done flow that is empty or contains only an ISC_NOOP. This is characteristic of a device that does not have the "done" bit included. Interrupted configuration of the device may therefore result in activating a partially programmed device on power-up.

If a device has all IOs defined as system pins then a more subtle deviation does not necessarily show mere compatibility. This means that these pins do not have the expected float or clamped behavior during ISC operations defined by the standard. A device of this sort is still fully compliant with the standard but system pin states may change during configuration. This puts the onus on system designers to take special care when designing in these devices to avoid creating potentially damaging conditions on the board during programming.

2.2 Power consumption during configuration

Although it is difficult to get the information about the power profile of in-system configurable devices from the device manufacturers, it is worth the effort to try to get some answers. In particular, it is important to understand if the device has any unusual power needs during erase, configuration or start-up.

Most manufacturers will provide you with information about the power consumption of their devices after configuration but few will provide you with information about the device power needs during ISC. Often, this is because such numbers are difficult to define usefully. They may also be difficult to predict, as they may be design dependent.

The typical volatile device does not have any unusual power needs during configuration. In fact, the device is likely to need less power during configuration than during normal system operation.

The same is not true of nonvolatile devices. These devices must typically enable various on-chip charge pumps during configuration. In addition, depending on the technology used and the techniques applied to effect erase and programming, there might be a high current need during one or both of the erase and program operations.

You should be acutely aware of the power needs of the selected device during configuration. If your system is power sensitive, you should explore the use of volatile programmable devices with a nonvolatile store. This store could be a separately powered system (a remote disk or other network store) or a flash memory with an interface suitable for configuring the volatile devices at power-up. However, before converting to a volatile device, you should engage in a careful examination of its power-up profile. It is not always the case that volatile devices use less power at start-up.

2.3 Configuration Speed

Depending on your system start-up time budget, you may need to examine the overall configuration time of each device. Typically, the configuration time consists of two parts. The first part is the time to shift in the configuration data (and any extra time to shift in the instructions and set-up information). This maximum configuration clock (TCK, usually) frequency typically fixes this time. The second part is the "burn" times associated with the various configuration operations like erase, program and read.

For nonvolatile devices, it is usually the case that burn times contribute the most to the overall configuration times. For volatile devices, it is usually the case that configuration data shift times are the key contributor.

Doing the configuration operations concurrently can mitigate the overall cost of configuring nonvolatile devices. In the ideal case, concurrent

configuration brings down the overall configuration time for many devices to the configuration time for a single device. This will only be the case if the configuration data shift time is at least ten times smaller than the burn times.

When using volatile devices, you must remember to consider the configuration time for the nonvolatile store associated with the devices.

When using nonvolatile devices, the configuration time is paid only at initial device programming or update. After that, the device will be functional from system power application.

2.4 Endurance

The power of in-system configuration lies within the ability to reconfigure the devices. *Endurance* is the number of erase and program cycles a device can withstand and still hold its configuration. This is typically an issue only for nonvolatile devices.

At a minimum, devices should allow 100 such cycles. Most devices available today offer many more erase and program cycles than that.

The endurance number is important for field-upgradeable systems. It contributes to estimation of the upper limit of the lifetime of the system. It also can help dictate the update strategy. For instance, if the endurance is large (say, 1000 cycles), it may be easier to reprogram all devices during an update even if only one of them changes. If the endurance number is small then each update may significantly reduce the lifespan of the system so only changed devices should be reprogrammed.

2.5 Data Retention

Clearly, the worst case is your system "forgetting" what it is doing during operation. Data retention is the measure of how long a device, once programmed, will keep its programmed data. This limit applies only to nonvolatile devices. While powered, volatile devices keep their program store. Therefore, in designing your system you need to be aware of the estimated product lifetime and choose device that matches it.

Nonvolatile devices should typically provide 10 years of data retention. It is usual for this value to be much longer.

2.6 Security

One of the security problems that volatile devices have that nonvolatile devices don't have is separate storage of the bit stream. A competitor could intercept the bit stream while it is being transferred to the target device and then put it into another device. Nonvolatile devices only have this issue when they are being remotely upgraded in the field since there is otherwise no bit stream to intercept. An available solution to this problem for some volatile devices is the use of encrypted bitstreams. In this situation, encrypted bitstream configuration data is delivered to the device. The keys for decrypting the configuration data are preprogrammed into the target FPGAs. The currently available solutions need a battery to keep the programmed encryption key alive even if the power goes out.

2.7 Reliability

Obviously, the selected device needs to be reliable. It needs to configure easily and correctly every time. During manufacturing configuration, it is usual for some devices to fail. Causes that contribute to configuration failure include surrounding electrical noise (for example, from testers or other equipment), device handling (for example, static discharge issues) and even improper device placement on your target board (for example, a rotated chip). This fallout, however, should be small; less than 0.5% is typical. Smaller values are also likely.

It is difficult to get reliability values from manufacturers since too many of the issues that cause fallout are outside their control. You will have to try to get this information chiefly from the experience of others in the field. Sometimes conference papers provide an idea of the typical manufacturing fallout of programmable devices in specific circumstances.

2.8 System Boot Time

The system boot time may include a significant part related to the configuration time. This depends on the total number of programmable devices and the use of concurrent configuration techniques. Simply stated: the whole is the sum of the parts.

The time to reboot may also be different depending on the shutdown sequence that preceded it. For instance, if the main-system board remained powered up but the end user station did not, then the boot time may consist only of restarting the system software.

Consider an application in which all the programmable devices are nonvolatile. The boot time consists only of the time needed to start the system software once nonvolatile device configuration completes. If the system shuts down normally, then the following boot time consists only of the system software start up time. Restart after a catastrophic shutdown, like a power outage, will likely need device configuration integrity check, followed by device reconfiguration, followed by a system software start-up.

Some systems may need to use a nonvolatile device to sequence and control the boot-up of volatile devices.

The portion of the boot time needed by configurable devices is an essential part of your start-up time budget.

2.9 Configuration Process Validation

You must consider the process that manufacturing and field personnel use to configure a system. With programming techniques, other than IEEE STD 1532, it is common for the device design data to be converted to several intermediate forms before use by the configuration tool. Depending on the circumstances (that is, prototyping, manufacturing, or field upgrade), different data formats may be used.

For example, a JEDEC file may be converted to SVF and then converted to a proprietary device programming language for performing configuration. To complicate the matters, this process may be different for each vendor of a configurable device that is on a board. Assuring that each operation is performed correctly and results in the correct configuration data being programmed in to the correct device can be a logistical challenge. This is further complicated when a design needs to be updated.

This is one major reason to select IEEE STD 1532 compliant devices. Configuration using the IEEE STD 1532 compliant vendor design files directly as input for the configuration procedure can simplify the flows significantly. The IEEE STD 1532 ISC data file also can contain CRC values to help in assuring and identifying versions.

3. Signal Layout Considerations

The device manufacturers specify configuration performance at the device pins. In-System Configuration tool vendors specify configuration performance at the connections to their systems. It is helpful to ensure that these two are compatible with one another. The following guidelines will help in the design of the layout of components, boards and systems.

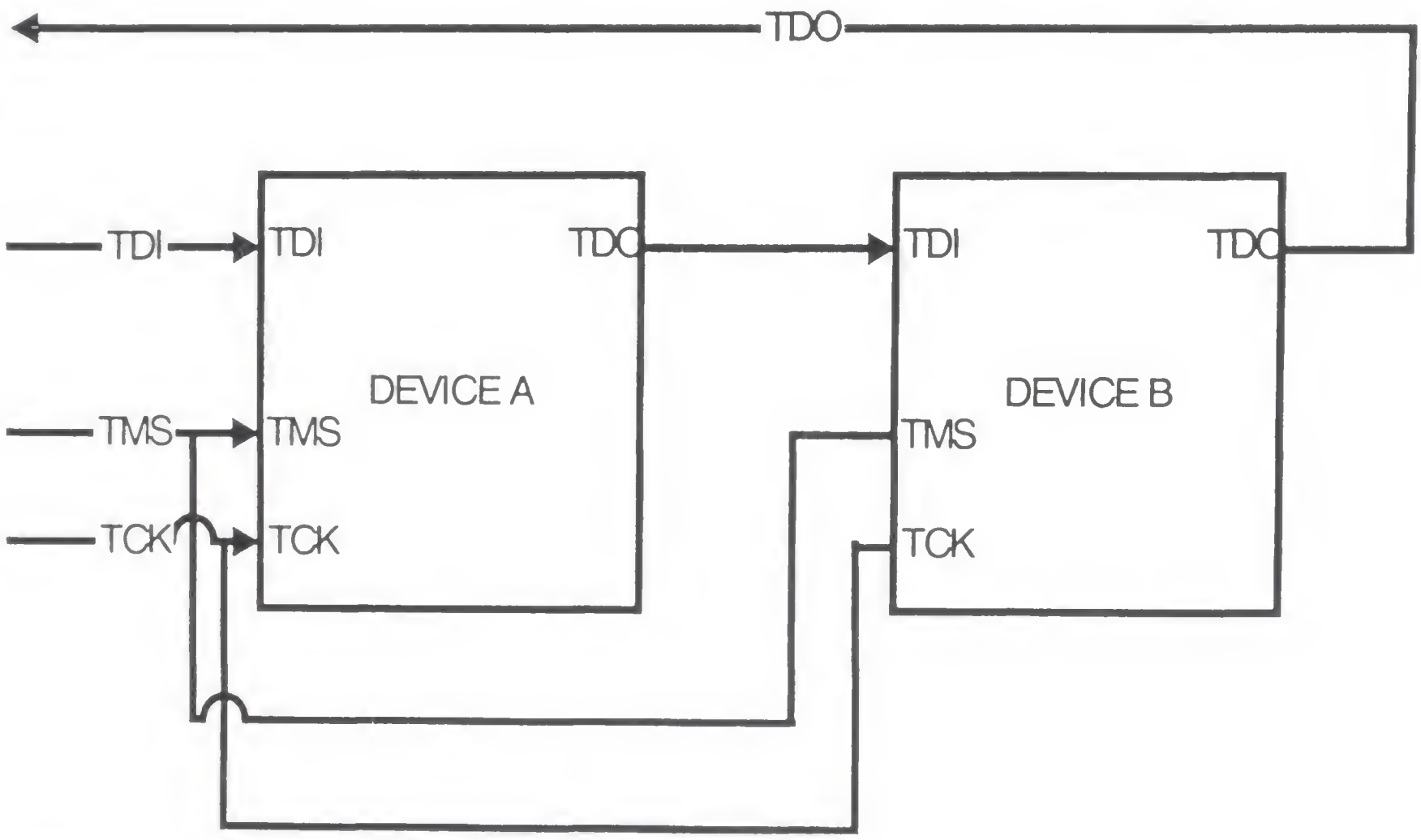

Figure 9-1. Serial Chain of IEEE STD 1149.1/1532 Compliant Devices

The four pins of the boundary-scan TAP consist of two serially connected signals and two parallel-distributed signals. The optional and rarely used fifth pin is TRST. The TRST pin activates an asynchronous TAP state machine reset. The TAP state immediately transitions to the Test-Logic-Reset state. One reason devices rarely have TRST is that driving TMS high for five pulses of TCK also resets the TAP state machine. Another reason is that, for reliability reasons, the designer may be needed to remove the possibility of transients accidentally resetting devices during operation. When present, you must connect all TRST signals together. Take special care to pull TRST high using a resistor during normal operation.

Wiring to the two serially connected signals, TDI and TDO, should be as short as possible. That is, connect each device TDO direct to the succeeding device's TDI with a minimum of extra routing. Even though the signals may be slow when compared with the system signal speeds, too much routing will cause excessive loading and could result in having these signals missing

the rising edge of TCK. Slow rising signals in CMOS circuitry are subject to noise and increased current consumption. These effects further degrade robustness.

The two parallel-distributed signals are TCK and TMS. TCK is a clock. Typical TCKs run in the range of several to several tens of megahertz. Once again, these signals may seem slow when compared to the system signals speeds. This, however, is not a reason for ignoring correct distribution and layout rules for these signals. Clock signal energy splitting and reflections can occur on badly laid out clock lines regardless of the operating frequency. This can cause sharp clock edges to become stair steps with ringing noise. When delivering TCK to more than 4 to 6 devices it is wise to use a clock tree to ensure delivering TCK edges is correct and synchronized at all devices. The TAP controller state machine is a synchronous state machine controlled by TCK and the correct operation of the entire chain relies on that fact that all devices are in the same state simultaneously.

Sampling the TMS signal occurs on the rising edge of TCK. The state of TMS then decides the TAP controller state machine's next state transition. TMS must arrive at all devices in time for TCK's rising edge to sample its state correctly. For these reasons, you should treat TMS like a slow clock. TMS therefore should use the same style distribution network as TCK.

The chosen clock-tree design, should seek to reduce clock skew and design clock buffers to meet skew specifications and lessen clock-tree power dissipation.

http://archives.e-insite.net/archives/ednmag/reg/1997/031497/cs_fg1.htm
http://archives.e-
insite.net/archives/ednmag/reg/1997/031497/cs_fg2.htmThe three most popular clock-tree implementations are:

1. The H tree
2. The Clock Grid
3. The Balanced Tree

Custom layouts use the H tree approach. In this approach, you vary the tree interconnect-segment widths to balance skew throughout the system. As drawn in Figure 9-2, black dots mark the clock drivers. The clock driver placement ensures the drive across the horizontal of each "H"-shaped route is balanced and correct for the segment. No clock driver powers more than two other clock drivers. Similarly, no clock driver connects to more than two clocked elements. The boxes in the diagram indicate the clocked elements.

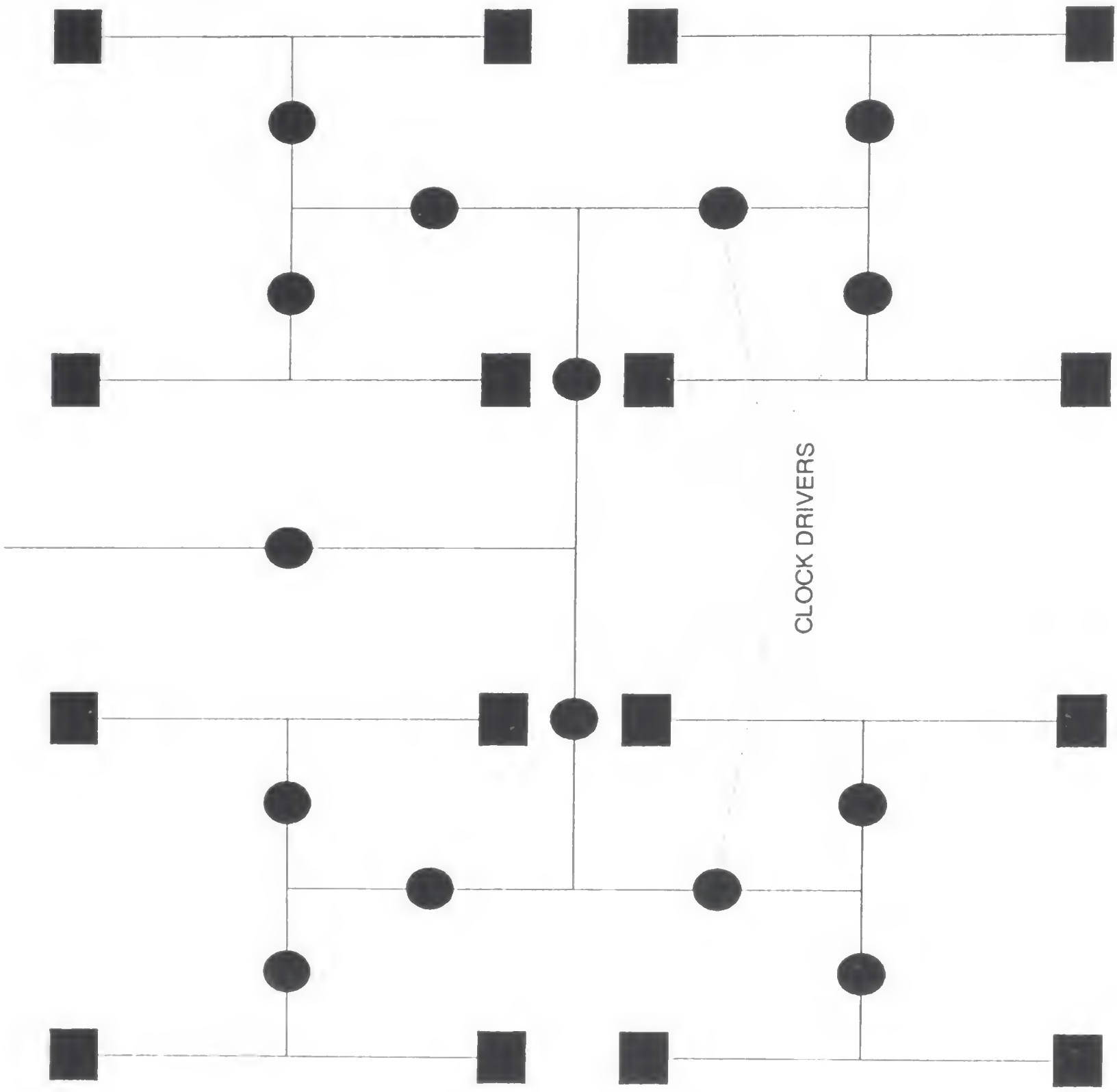

Figure 9-2. The H Tree

The clock grid is the simplest clock-distribution network and has the advantage of being easy to design for low skew. However, it is area-inefficient and, even worse, power-hungry because of the large amount of clock interconnect it needs.

As shown in Figure 9-3, the clock grid overlays the system with a grid to allow wide distribution of the clock signal. Large clock drivers (as indicated by the black dots) are arranged across the grid. Not shown in the diagram is the manner in which the grid is supplied the clock. Typically the clock is driven into the center of the grid by a single source. Depending on the size of the system grid, it may be necessary to have a secondary or tertiary grids to distribute the clock to the system grid. The secondary and tertiary grids look like H trees overlaying the grid. It is clear and obvious how this can quickly become a power and area hungry solution.

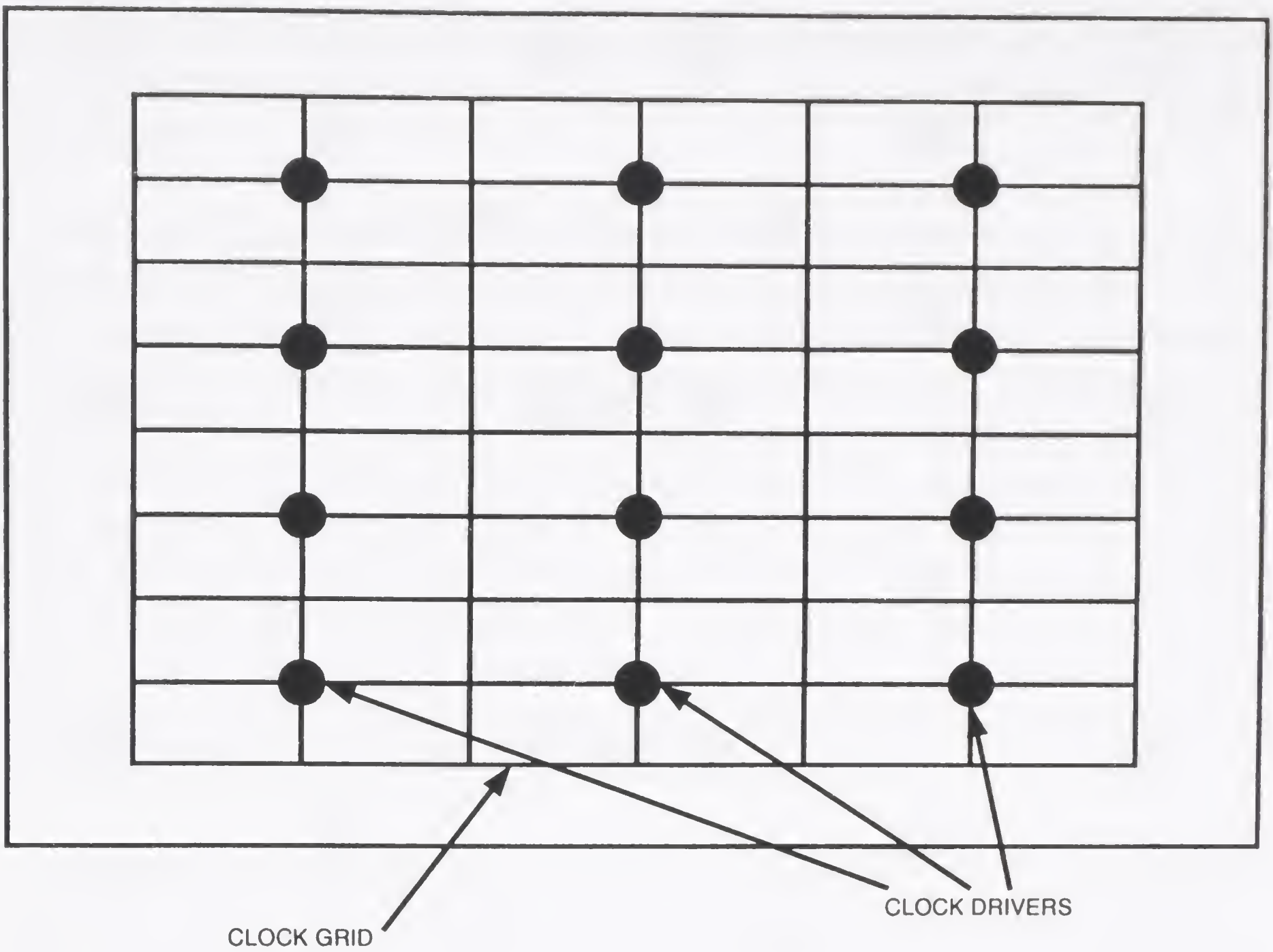

Figure 9-3. The Clock Grid

The balanced tree is the most common clock-distribution network. It is shown in Figure 9-4. A balanced tree without buffers is one in which the clock lines' capacitance increases exponentially as you move from the leaf cell (clocked element) to the root of the tree (clock input). The extra capacitance results from the wider metal needed to carry current to the branching segments. The extra routing also results in added area to house the extra clock-line width. Adding buffers at the branching points of the tree (depicted by dots in the Figure) significantly lowers clock-interconnect capacitance, because you can reduce clock-line width toward the root.

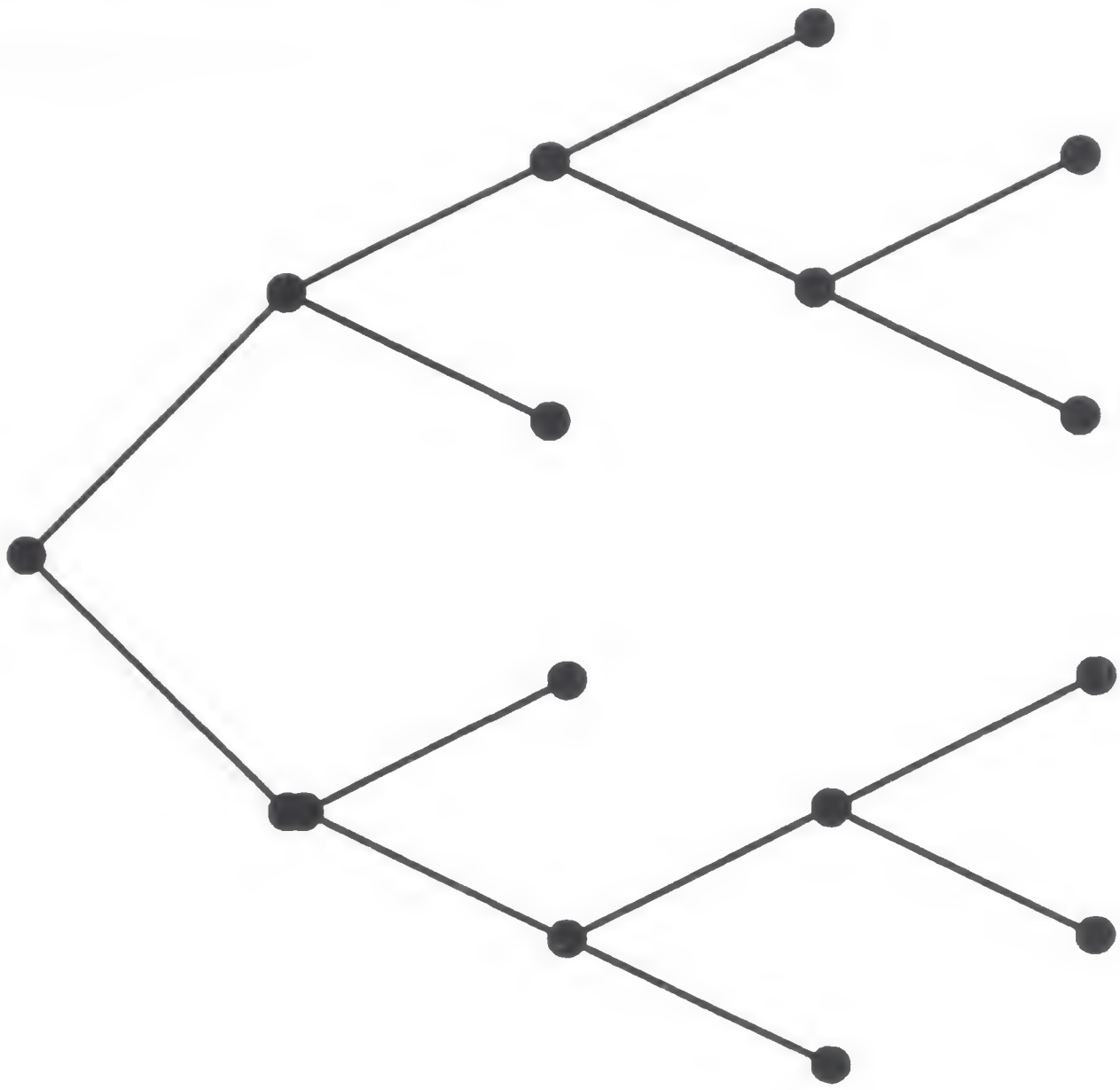

Figure 9-4. The Balanced Tree

4. System Power Considerations

Depending on whether you configure one device at a time or are using concurrent programming to speed system configuration you will need to ensure that your system power supply is able to provide the necessary device power. In particular, programming a single device at a time rarely strains a system power supply but programming a group of devices concurrently may exceed the capacity of your system power supply. Consider the condition in which devices begin functioning in mission mode immediately on completion of configuration. Suppose a single device has a high current need for configuration, then the sum total of the configuration current need and the mission mode current requirement may exceed the maximum capacity of the system power supply. In addition, sudden large configuration power demands may need special board ground impedance design to handle the current spikes to avoid spurious noise.

You might be able to use a special external supply during manufacturing to provide the necessary current. But, if you are planning on updating your system in the field, the system power supply will have to perform to the

specifications associated with providing the peak current needed to configure a group of devices concurrently.

Another alternative that does not need a larger power supply is not to allow execution of field upgrades using concurrent operations. This approach might increase system downtime during configuration updates, but it reduces the power requirement.

A further constraining alternative is to configure devices one at a time and to hold back starting up devices after configuration. Then start them only after all devices have configured. This guarantees the power needed is always less than the system need when running in mission mode.

If during power-up, the system powers the buffers controlling TCK, TMS and TDI after the target circuit then extra transitions on these signals may be produced. This can initialize the TAP state machine in an unexpected state. One way to correct this is to begin all TAP operations with a TMS controlled transition to Test-Logic-Reset.

5. Device and System Test Considerations

You should also carefully consider the ability of the device to complement your system test needs. If you intend to perform interconnect test using boundary-scan you will want to make certain that devices are fully IEEE STD 1532 compliant since they are then also fully IEEE STD 1149.1 compliant. If you are planning on doing some device testing as well, you will want to make certain the selected devices support the IEEE STD 1149.1 INTEST or RUNBIST instructions or some proprietary equivalent.

It also possible to use the EXTEST instruction coupled with the boundary-scan register of some devices to program commodity flash devices that are not strictly in-system configurable. Some IEEE STD 1532 devices allow you to interleave these EXTEST operations with their own ISC operations. This interleaving of work can allow for faster overall system configuration times. Devices that allow interleaving of EXTEST with ISC operations will not have the ISC_ILLEGAL_EXIT attribute defined, or if defined, its instruction list will not include EXTEST.

Sophisticated test applications may also be able to interleave interconnect test and configuration actions.

You might plan to use the programmable devices in your system as test and diagnostic hardware at power up. To do this they must reconfigure to their mission mode function after successful test completion. You will also need to ensure the test and mission roles are complementary and don't make different and conflicting demands of the system under test that would make this dual functionality impossible to realize.

6. System Configurability Considerations

What class of configurability does your system design include?

1. Prototyping Configuration
2. Production Configuration
3. Field Upgradeable
4. Bi-configurable (at boot time, diagnostic and test and then mission mode)
5. Functionally Reconfigurable at run time
6. Medley Reconfigurability

In this section, we will examine the system design considerations for each of the mentioned classes of configurability. We will examine these by examples in Chapter 7.

6.1 Prototyping Configuration

This describes a system configured only during system development and prototyping. This means allowing rapid access to configurable devices to reconfigure each with new designs as bugs are found and fixed. It is also likely the configuration port may be used for debug access. There may be no later reconfiguration need.

A system of this sort will likely have a port available for configuring the devices on the system. This port will be on the system printed circuit board and may be a separate connector. This port, however, may not be accessible after the prototyping as the connection hardware may be removed from the board for production.

The design of the system will not necessarily involve anything more than making certain that all necessary configuration control signals are available at the board edge. The signals are made available by a connector which

mates easily with the system performing the configuration. This could either be a programming cable or a stand-alone programming station.

6.2 Production Configuration

This describes a system configured only once during manufacturing. This might mean loading the system with geographically selected or feature set limited programming patterns. A part of the production flow is setting the exact system configuration. The configuration never changes afterwards. Therefore, there is no later reconfiguration need and the system design does not allow it.

A system of this sort will likely have a port available for configuring the devices on the system. This port will be on the system printed circuit board and may be a separate connector. However, this port may not be accessible after the product packaging.

The design of the system will not necessarily involve anything more than making certain that all necessary configuration control signals are available at the board edge. The signals are made available by a connector that mates easily with the system performing the configuration. This could either be automatic test equipment, a programming cable or a stand-alone programming station.

6.3 Field Upgradeable

The field upgradeable class of reconfigurable systems describes a manufacturing-time configured system whose design allows for irregular updates after placement in the field. The expectation is there will be limited updates, perhaps performed once or twice a year at the most.

The nature of the expected field upgrades is important to understand. For instance, is the expectation that a service engineer will carry out upgrades in the field? Will the service engineer have sophisticated equipment equivalent to a laptop PC? Alternatively, will the service engineer only have a handheld system of limited functionality? On the other hand, will they only have an upgrade disk? Will there be a wireless interface to the system?

Will access be limited to the four pins of the IEEE STD 1149.1 TAP? If so, the designer may need to ensure the TAP can control that certain non-boundary-scan devices if they interfere with system configuration.

Will a central office carry out the field upgrade? If so, is a service engineer expected to be present at the site?

The simplest approach to field upgrade is to reload all device configuration files regardless of change when any one needs to be updated. This makes the procedure simple but may demand added endurance from the devices since the total number of erase and program cycles will be equal to the total number of changes expected to all device contents in the system.

These variations on the theme of field upgradeability have direct ramifications on the design of the system.

6.3.1 Field Upgradeable – Service Engineer

The presence of a service engineer equipped with a PC means the system need not have its own configuration controller. In fact, this system is a variation on the production-configured version above except the configuration port needs to be accessible to the service engineer. The easiest way to do this is to make the port accessible by a service door on the system.

6.3.2 Field Upgradeable – Remote Control

If the system needs remote upgrade, there will have to be a processor available to act as a configuration controller and manage the configuration data reception.

This may be a dedicated processor or it may merely be a service function of the main processor. Since the role is unlikely to used often, if you are using a dedicated processor, it ought to be an inexpensive processor.

Another possible variation is to use a processor to manage the communications with the remote site and a dedicated core or assembled logic to read the configuration data out of a memory store and apply it to the devices. The memory store could be a flash memory or other suitably sized nonvolatile store.

With remote control field upgrade, it is usual to introduce the new configuration information in a stepwise fashion to ensure there is always a working back-up configuration available. This means that two memory banks need to be available. The first is the active bank. This bank stores the configuration data used to program the target system. The second is the auxiliary bank. This receives the updated configuration data. After data

receipt, the configuration controller confirms the data integrity typically through use of a CRC check or checksum. The system then uses the auxiliary data bank as the configuration data source bank. If there is any problem detected in receiving the new configuration data, the system sends a message to the central office. The active bank then remains unchanged. If configuration controller confirms the updated configuration data as correct then it configures the system using the new data. When the configuration controller corroborates that system configured correctly (and potentially tests that it works correctly), it sets the auxiliary bank to be the active bank. It then sets the previous active bank as the auxiliary bank and erases it. It may also choose simply to leave the previous active bank contents untouched. If the system either did not configure correctly or did not work correctly after the update, the active bank remains as the system function and configuration controller sends a message to the central office.

The frequency of reconfiguration of field upgradeable systems may be as often as once every three to six months or as rarely as once every few years. It may also be the case, that reconfiguration of field upgradeable systems occur more often early in their life cycle and more rarely later in their life cycle.

6.4 Bi-Configurable

This bi-configurable style of system is one that has two distinct configuration phases on power-up. The first is setting the system to perform self-test and potentially diagnostic functions. When completed successfully, the system reconfigures itself to perform its intended mission function.

A central office usually oversees this system. Ideally when the diagnostic fails, the system alerts the central office to schedule repairs.

A system of this sort has two sequentially activated configuration images. The first is the diagnostic image; the second is the mission image.

The system loads each image, one at a time. The system loads the diagnostic image and runs it. The system reports failures either to a console or directly to the central office. If the diagnostic passes, the system loads the mission image and starts.

This system needs a configuration controller but it need not be a microprocessor. A simple sequencing state machine may be enough to step through and perform the necessary operations.

At boot time or power-up, there will be two reconfigurations of bi-configurable systems. One configuration will set up the diagnostic role and the second will set up the mission role. These reconfiguration steps will each occur one after another and the interval between the two configuration steps will be just a few minutes. After that, no further reconfiguration will take place.

Since bi-configurable systems may perform diagnostic or other debug functions, available test functionality, including IEEE STD 1149.1 compliance, may be critical.

6.5 Functionally Reconfigurable

In a functionally reconfigurable system, configurability is an essential part of the system functionality. As the system runs, it is either constantly or at regularly reconfiguring itself as a part of normal operation.

An example of this class of system might a digital signal filter that constantly adjusts itself according to the conditions of the input signal.

Some systems of this sort reconfigure themselves by watching input signal data and output signal data. Then, based on the needed accuracy and shape of the output signal, an algorithm calculates the new configuration memory contents and then streams the result directly to the configurable device. In this manner, the system is constantly adjusting itself to produce better output. Systems of this sort are also known as dynamically reconfigurable systems.

Researchers developed systems that use techniques like this to learn what to do and dynamically adjust their configuration until they converge on the wanted functionality.

Another example of this system is one that uses a central processor to watch the data processed. Then based on the data passing through the device, the processor selects from various available configurations in a library and then configures the device (or a portion of it) to handle the data correctly.

You could imagine a communications switch built using this approach that watches the input signal data for specific communications protocols. On identifying a new incoming protocol, the system reconfigures on the fly to handle the incoming data properly.

Yet another example of functional reconfigurability, is using it to provide added system security. In this circumstance, a central office upgrades systems often to accept new protocols or security keys to deter piracy.

One can imagine many other variations of this class of application. What is common to all is the system must reconfigure itself during operation to work correctly. The frequency of reconfiguration is as high as few milliseconds or as low as every few hours.

6.6 Medley Reconfigurable

Systems that are "medley" reconfigurable take a little sample of each of the variations and mix them according to needs. It is possible, for instance, to develop a system that is Bi-Configurable **and** field upgradeable or a system that is functionally reconfigurable **but only** production configured.

This allows for developing systems that take some of the features and benefits of each variation by delivering a new reconfigurable system.

An example of a medley reconfigurable system is an application that stores separate functional configurations, all of which are programmed during production. Some time after production a technician sets up the needed configuration, perhaps by setting jumpers or switches just before product ship. Consider a VCR that targets North America, Europe and Asia. At production time, the electronics for processing each geographic locale is programmed into the system. It will never be upgraded so the design incorporates production configuration only. Just before the VCR ships to its end-market, a technician sets some switches or jumpers to adjust the system application. This makes it field upgradeable as well. In this case, though, it is upgradeable once and from only a finite set of choices.

7. Summary

In this chapter, we introduced the essentials of reconfigurable devices and systems. In the following chapters, we will build on this information to see how existing applications and tool sets use this configurability. Then we will further examine how these tools and techniques can help simplify developing the variety of reconfigurable systems described.

Chapter 10

IN-SYSTEM CONFIGURATION-BASED PLATFORMS

1. Configuration Environments

Configuration of a system has a life cycle like any other entity. A configurable system is likely to exist in many different environments during its design, development, manufacturing and deployment. Each of these environments pose significant and often different and contradictory sets of expectations and feature needs.

In this section, we will examine the variety of operating environments and the applications available to support them as well as the demands made on the configurable system itself.

1.1 Prototype

The prototyping environment is the product development and debug environment. During prototyping, rapid and almost continuous reconfiguration is the norm. The work environment is a laboratory or a workbench. The developer is likely working in an electrically noisy and chaotic environment. Designers expect problems (including configuration problems) during development and debug but obviously, they prefer error-free and successful execution. As development continues, problems related to configuration should disappear. As a prototype, the system may have many jumpers and may need changes to make it operable. Changes include cutting traces on the printed circuit board, adding capacitors and resistors to nets and adjustments.

While prototyping, the designer is in control. The designer focuses on getting the system to work. After the system works, the designer will have to revisit the adjustments made and decide how to incorporate them in the final design.

A typical designer focuses only in configuring the devices that form the portion of the system for which they are responsible. A simple configuration tool running on a PC and using a vendor-supplied download cable would probably be enough.

1.2 Manufacturing

The manufacturing environment needs the system to have guaranteed configurability. The configuration data must be stable and available. Configuration should work first time and every time. There must be no fallout owing to configuration problems. There is some small amount of adjustment allowed to the environment to make up for limitations. This might include slightly higher supply voltages, supplies that are more powerful, more noise resistant cabling and the like. These adjustments, however, may be problematic if the system is field upgradeable since the field-operating environment may not allow for these compensatory measures.

Product engineering is in charge during manufacturing. The product engineer focuses on making sure the yields are high and the throughput (that is, the number of units produced and tested each hour) maximized. In other words, minimizing the production cost.

System assembly occurs in the manufacturing stage. This system may contain devices from many different manufacturers. All the devices will need configuration. Device configuration needs a reliable tool set that is immune to production floor noise. Obviously, manufacturing wants support for configuration of all manufacturers' devices. Simplicity is important for production to be efficient. Cables used to download device configurations; these cables should support all devices. You do not want to have a condition in which production must stop to change cabling to configure a new device.

It is also the case that during production, use of a single integrated configuration step increases overall throughput. This suggests running integrated test and configuration steps on automatic test equipment (ATE). Ideally, the integrated approach should yield a single file or program that carries out all test and configuration operations.

1.3 Field

When releasing a system to the field with the likelihood of eventual field upgrades high, the upgrades must occur without failure. The configuration environment is the run-time environment of the system. There is no possibility of changing this environment or making up for any limitations or irregularities.

This means designing the system to ensure that upgrades are possible and reliable. This might be in conflict with minimizing the production cost. This will be offset though by the extended lifetime of the system and the reduced maintenance costs associated with the product.

2. PLD Manufacturer Tools

Every programmable device manufacturer provides some application software for device configuration. PLD manufacturers typically provide this software free. A PC version is always available. The applications are also often available for UNIX workstations and Linux support is now emerging. Users must however buy a specially designed download cable to configure devices. These cables typically retail for between $100 and $500. Download cables are available to connect to a PC parallel and USB port as well as serial ports.

Device manufacturer supplied download cables are incompatible with one another. This means that one manufacturer's download cable is not usable with another manufacturer's download application. It is also true that a jig designed to connect to one manufacturer's cable pin out will not correctly connect to another manufacturer's cable. Please note that manufacturers do provide flying lead connections for their cable heads but this makes the cable to target connection mechanism more difficult.

In addition, unless an application supports IEEE STD 1532-based configuration, you will need to use a different application for each different manufacturer's device used in the design. In the worst case, this means that several different applications, download cables and cable-to-system connections will need to coexist if you are using multiple manufacturer devices.

Manufacturer-provided solutions are likely to be ill suited for use in a manufacturing environment. The design of the download cables is rarely

able to withstand the needed number of reconnections and strain leading to the cable simply wearing out. There may also be a significant susceptibility to surrounding electrical noise.

For use as a field upgrade tool, manufacturer-supplied applications need a PC and the associated download cable. This might not be suitable if you do not have system access to the cable connection point or if taking a PC to the field location is not possible.

Manufacturer tools are typically your only source for the generation of intermediate file formats like SVF or STAPL. If you are using some third-party systems like certain PC-based boundary-scan tools or ATEs for device programming, you may need to use SVF or STAPL as input to describe the configuration algorithm and data. If these systems do not accept IEEE STD 1532 BSDL and ISC data files then typically one of these alternative formats are supported.

2.1 PLD Manufacturer Specialty Tools

In the time before IEEE STD 1532 and without a standard, some vendors proposed their own approaches to solve end user needs. There was a vital need for an efficient embedded system programming solution. While IEEE STD 1532 makes these approaches obsolete, there is a large installed base. Some end users will continue to use these approaches in existing systems.

2.1.1 Xilinx XSVF

As already noted, SVF files have two basic weaknesses. First, they can get large because the data is represented in ASCII HEX. Second, they cannot represent algorithmic control flow.

These two issues loomed large for Xilinx when they needed to provide a configuration solution suitable for use in embedded systems. To resolve this problem, Xilinx created what was essentially a binary encoded version of SVF called XSVF (presumably to mean Xilinx SVF). The format took the basic SVF command set and encoded the data in binary. As well, several Xilinx-specific extensions were included that optimized the data representation. This included a record that understood the address format of Xilinx's CPLDs to allow for software increment, a record that understood the device's return status to enable a retry mechanism and built-in assumptions about state transitions. To support this, Xilinx provided a

translator that took SVF files for Xilinx devices and translated them into XSVF. Xilinx also provided a small C code interpreter of XSVF files. This interpreter was less than 10 kilobytes in size after compilation and proved to be useful in embedded systems.

The approach provide to be popular and the applications note and its associated software quickly became one of the most popular downloads from Xilinx's web site. Users could configure Xilinx-based systems (bypassing other vendor devices) using TAP access and a simple microprocessor. There was a limited run time overhead and shift speed limited only by the processor used to drive the TAP pins.

The solution is tuned and tested with Xilinx devices only (CPLDs, FPGAs and PROMs). Although technically, simple SVF files produced by manufacturers other than Xilinx should be usable in this flow, there has been little publicized effort to prove or disprove this theory. Therefore, for practical purposes this solution remains applicable only to Xilinx devices.

2.1.2 Lattice Semiconductor ispVM

After the introduction of STAPL, the development of the JAPIBS and while IEEE STD 1532 was still in the definition stages, Lattice Semiconductor proposed a new approach. This approach sought to combine the best characteristics of STAPL and JAPIBS and support IEEE STD 1532 when completed.

The basic idea was to build a solution using virtual machine technology. Rather than employ the Java virtual machine that included general purpose computing support, a smaller one could be realized by limiting the requirements to simpler operations of in-system configuration. This became the ispVM.

The entire ispVM assumes that device algorithms are described completely and effectively in SVF. The SVF description is then compiled into a byte code format. The byte code can then be run on any ispVM implementation. Alternatively, applications that are ispVM byte code "aware" can produce the device programming algorithm in ispVM byte code directly.

Lattice Semiconductor built two significant applications based on ispVM technology. One is the ispVM System Software. This provides a complete environment in which to execute and debug SVF files (after their translation to ispVM byte code). It also accepts IEEE STD 1532 BSDL and ISC data files (that adhere to the 2001 version of the standard) and allows direct execution. It does this by first compiling them to byte code.

The drawback of this approach is that some devices do not have algorithms that can be described in SVF (as we have already discussed). In addition, with the approval of the 2002 version of IEE STD 1532, devices with non-deterministic configuration algorithms can be described. Devices of this class cannot be supported in ispVM as it is defined.

A second serious drawback is the details of the ispVM byte code format are not available to the public. This means that independent implementations of the ispVM cannot be developed. It also precludes generation of optimized byte code descriptions of algorithms. The only path to ispVM byte code is SVF files compiled by Lattice's tool set.

2.2 PC-based Boundary-Scan Tools

As the need for stand-alone, low cost boundary-scan test and debug stations increased, several suppliers arrived on the scene. These suppliers developed applications that use their own algorithms and hardware to perform IEEE STD 1149.1-based system test and debug.

When IEEE STD 1532 began to expand, these same vendors extended their IEEE STD 1149.1 support to include in-system configuration based on IEEE STD 1532. By integrating in-system configuration solutions with their boundary-scan tools, these vendors provide an in-system configuration, test and debug solution. They usually make available "in-system configuration-only" applications as well.

These applications support all manufacturers' devices through a single download cable (designed and made by the application developer). Most application developers also have TAP controller modules in various form factors from PC plug-in cards, to VXI bus cards, to USB and Ethernet interface modules. These provide much design flexibility and portability.

Unfortunately, these systems and their associated hardware are not free. They can cost several thousand dollars for each license. These solutions do however provide a single source, multi manufacturer solution with high

throughput suitable for a manufacturing environment. If you already own a system like this for manufacturing test, then the only cost is the incremental cost of buying the configuration software.

To complete the configuration solution, most vendors offer add-ons that will configure on-board flash memory using the boundary-scan registers of surrounding devices.

This latter feature coupled with IEEE STD 1149.1 test, IEEE STD 1532 in-system configuration and debug may provide a satisfactory manufacturing solution. Where access to nets internal to the board is needed to precondition the board for boundary scan operations, the PC based boundary-scan tools may not be acceptable. Because these applications need nothing more than a PC to work, designers can use them during prototyping. Once again, this solution is not suitable in the field without system access to the cable connection point or if taking a PC to the field location is not possible.

3. Automatic Board Test Equipment Tools

There are several suppliers of in-circuit and automatic board test equipment, also known as ATE. This specialty equipment does loaded board testing of electronic systems. Loaded boards are boards populated with parts. Most ATE support IEEE STD 1149.1-based testing of these target systems. Similar to PC-based boundary-scan tools, ATE manufacturers have also extended their IEEE STD 1149.1 tool sets to include IEEE STD 1532 support.

Certain challenges that exist for IEEE STD 1149.1 support are, sometimes, more acute when supporting IEEE STD 1532. The primary issue is that of tester memory. ATE works by streaming vectors stored in memory associated with each pin on the tester, to an associated pin on the board under test. In many systems, this memory is the pin memory. One or more bits represent the state of stimulus driven from the pin memory into the board under test at any instance in time. Test programs usually have stimuli spread evenly across all pins resulting in rather balanced use of ATE memory. ATEs assign pin memory evenly across the test head. This arrangement is functionally depicted in Figure 10-1. The typical ATE vector memory is termed to be wide and shallow. This allows it to service many pins and stream a few thousand vectors to the board under test. This is typically sufficient to complete functional test of a complex system.

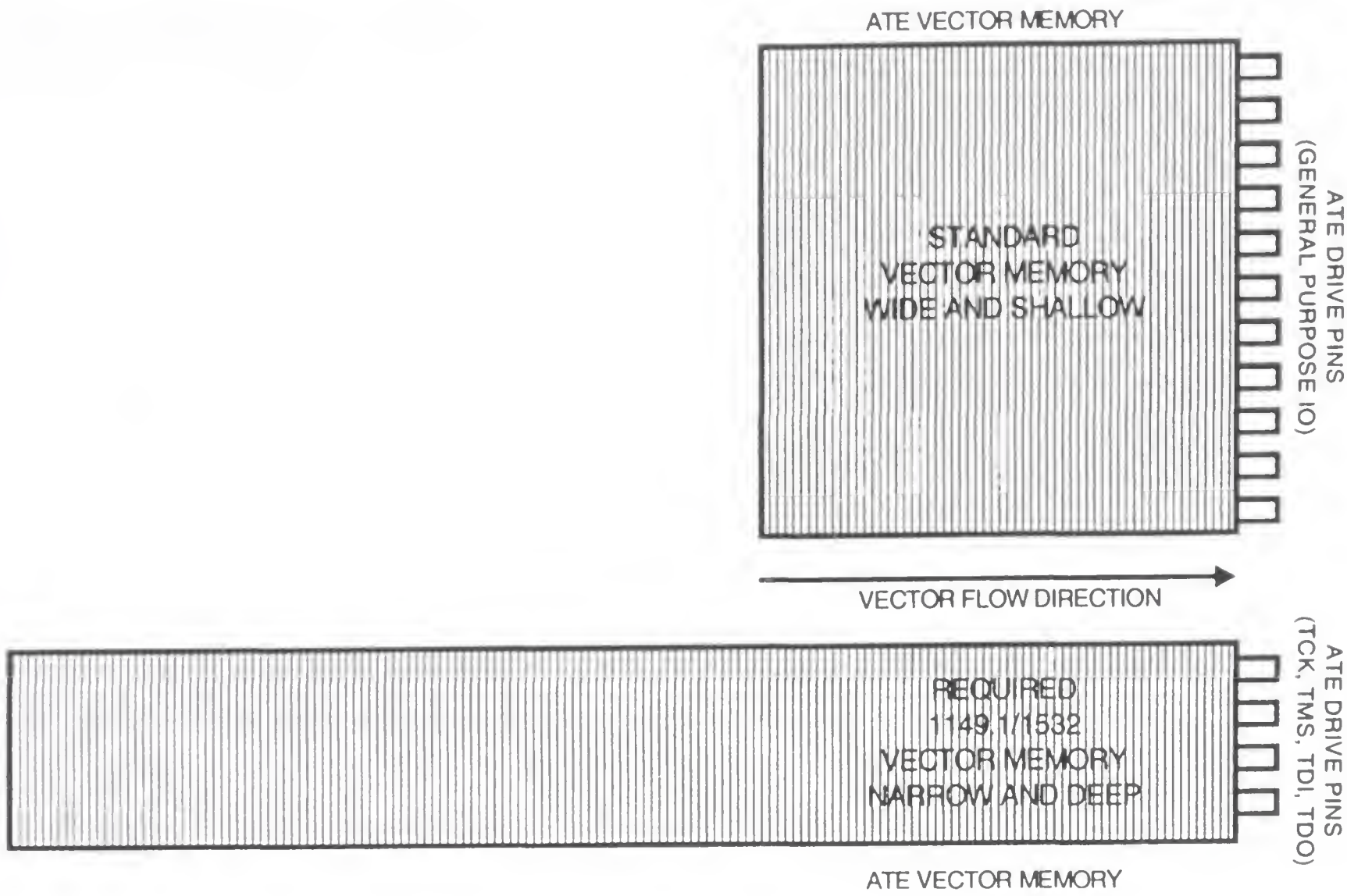

Figure 10-1. Memory Usage in Automated Test Equipment

Boundary-scan test programs need lop-sided memory consumption. Rather than a wide and shallow memory they need it narrow and deep. The three input pins of the TAP controller (TCK, TMS, TDI) consume most of the ATE memory. Since boundary-scan and configuration operations involve serially shifting large amounts of data, it is not unusual for the TDI memory requirement to be millions of bits deep.

This means that when using ATE for boundary-scan tests, you may need extra memory or added special hardware. When doing in-system configuration on ATE, the memory needs are even more significant since configuration sequences are usually substantially longer than boundary-scan tests. If you have several devices to configure, the memory needed may be prohibitively large. It is true that for a test or configuration program to work the entire sequence does not need to be resident in pin memory always. ATE allows storage of portions of sequences on disk for retrieval when needed. This caching approach becomes impracticable quickly if you must perform many disk retrievals. Disk accesses take a long time and can increase the overall ATE program execution time.

To overcome the pin memory limits, some ATE systems have integrated specialized Boundary-Scan Controller hardware. This hardware efficiently

exercises the TAP pins delivering the long serial streams of shift data needed to perform the boundary scan operations. This usually results in higher speed shift operations and better memory management.

ATE can offer rapid execution times while integrating in-system configuration and test programs when the pin memory issues are resolved. Similar to PC based boundary-scan applications; ATE can configure PLDs from multiple manufacturers. Some vendors provide PLD configuration applications as add-ons, available at extra cost.

Obviously, manufacturing environments are the primary hosts of ATE solutions. They can offer seamless integration of in-system configuration and test and provide this support without added fixturing. ATE also provides better access to other board pins. You may need to drive some board pins during test or in-system configuration to ensure success. This might include driving pins to disable active devices or clocks, to float bus signals or set certain control signals or state machines to safe states. An application that provides access only to the TAP will not be able to drive more pins. Owing to ATE cost and size, it is unsuitable for use in prototyping or in the field.

4. Field Application Tools

There are two possible approaches for performing in-system configuration in the field:

1. Direct TAP Access Method
2. Embedded In-system Configuration Processor Method

Performing in-system configuration in the field does not demand the throughput speed that the manufacturing environment does, so the use of the TAP port running at less than maximum speed is acceptable.

However, system security must not be compromised. You must ensure unauthorized people cannot tamper with the system. You do not want to either lose the system's intellectual property or have its function altered or removed. You can provide this security either physically or programmatically. Physical security techniques involve making the TAP access difficult. This might include needing the system to be powered down and dismantled, or having jumpers that need to be removed with special tools before accessing the TAP. Programmatic security includes such well-

known techniques as password protection, security keys or biometric interfaces.

The update technique must be foolproof. You need a mechanism to make sure the correct data is programmed in to each device and in the correct sequence. This may simply need a more sophisticated update application that can verify the update sequence was completed correctly before allowing the system to return to operation.

4.1 Direct TAP Access Methods

If you have access to a laptop PC and your system provides easy access to the TAP, then you have two choices. First, you could use the manufacturer-supplied application and download cable to connect to the system in the field. If you use many different manufacturers' devices in your system this becomes difficult to manage. Second, you could use a PC-based boundary-scan tool. This can more easily manage multiple manufacturer situations. The drawback remains the use of the laptop and its associated cable. In addition, it may not be desirable to have the TAP port easily accessible for security reasons.

4.2 Embedded In-System Configuration Processor Methods

The usual technique to provide field accessibility is to design a tethered configuration controller into your system. This means that you must embed the configuration data and algorithm into your system. As described earlier, the use of IEEE STD 1532 data and algorithm files provides an excellent approach for realizing this. Either the configuration data or the algorithm can be independently accessed for update or modification. You are free to select a configuration data compression algorithm suitable for your environment and data (should one need it). In addition, all devices are supported directly without needed extra translation steps.

Xilinx provides a free IEEE STD 1532 configuration environment called JDrive (available for download from the Xilinx web site – http://www.xilinx.com). Source code is provided that can be used to read and execute all conforming IEEE STD 1532 BSDL and ISC data files regardless of the source. This application is suitable for embedding into microcontrollers. Since source code is provided, developers can adapt and improve as needed in their target system.

Other approaches exist but are less satisfactory. Developing customized solutions based on proprietary approaches and data representations are possible. The drawbacks of such approaches are obvious. A continued effort is needed to maintain the solution and improve it to support new or added manufacturer devices. There may also be logistical difficulties related to coordinating and updating data. A key question directed at this approach is, "why reinvent the wheel?"

You might consider translation-based solutions like SVF or STAPL. Either approach is embeddable. Source code for a STAPL interpreter is available from the Altera web site (https://www.altera.com/support/software/download/programming/jam/jam-index.jsp). There are, as previously noted, certain weaknesses of STAPL. These weaknesses make it less suitable for general field applications. These include, the hard-coded compression algorithm, the large run-time memory need and weak separation of configuration data and algorithm.

There are no publicly available interpreters for SVF although construction of one is straightforward. SVF, in turn, has significant failings. The key issues, as previously described, are twofold. The file size is large since it represents binary data as ACSII hex characters. The configuration data and algorithm are tightly interwoven in the file.

A quick review of Chapter 4 will remind you of the details of each approach.

Chapter 11

DESIGNING IN-SYSTEM CONFIGURABLE APPLICATIONS

1. The Spectrum of Configurability

Each configurable system has an intended frequency of configuration. Some configurable systems are configurable only once – at manufacturing time. Others incorporate configurability as an essential system function and must be run-time configurable. The following categories define the spectrum of configurability.

Simple configuration may be the most recognizable configurable application. Technicians populated a printed circuit board with programmable devices. Part of the board test procedure includes programming the PLD's configuration memory with logic patterns. The system configuration is never again changed.

Field reconfiguration is not time dependent, but is the most pervasive application. It can provide a technique for updates, bug fixes, and adding new features to digital hardware.

Periodic reconfiguration is for those applications where there is a regular change of supporting data, such as environmental recording systems, global positioning systems, and so on.

Frequent reconfiguration speeds up data processing for applications such as image processing.

Runtime reconfiguration applications rely on changing working environment conditions. As these conditions change, so must the application function. Examples of this are detecting network protocols in a mobile application or interrupting a task with new service triggered by a security or safety sensor.

Some variations within this spectrum include **partial reconfiguration** and **on-the-fly reconfiguration.** These two variations can exist within the spectrum as a subclass of any of the described categories. They may also exist as a separate application class within the spectrum.

Partial reconfiguration allows reconfiguration of only a portion of the programmable device. This allows development of systems that can remain *mostly operational* during reconfiguration. Reconfiguration disables only the portion reconfigured. When used correctly, partial reconfiguration allows phased-in reconfiguration of features and tasks with limited impact on system utility.

On-the-fly reconfiguration is typically available only in nonvolatile devices. It allows programming of the static configuration memory of a device with an alternate utility. Activation of the alternate utility does not immediately occur. Activation occurs when a designer-specified trigger condition occurs. This allows development of systems that can preconfigure while running without disturbing the run-time behavior. When enabled, the alternate system utility activates with almost no down time. The total down time is equal to the time it takes to load the static configuration memory into the active memory (typically 100 microseconds).

2. Designing for Simple Configurability

What does a simple configurable system look like? The real problem is how to design a configurable system. In speaking of configurable, we mean configurable once at development or manufacturing time only. There are several basic rules of thumb associated with the successful design of configurable systems.

1. Design in configuration port accessibility.

   You need access to the port. If you can't, you obviously won't be able to configure your system even once. This also includes making certain that any port enable signals are correctly activated. In addition, if some devices support TRST and others don't, you must ensure that the stray TRST signals are appropriately controlled during configuration to prevent stray TAP transitions.

2. Design the configuration port interconnect network to function at the needed speeds.

Don't skimp on common sense design practices even if port use is limited. It relays the information that describes the system functionality. You want it to work and work reliably.

3. Make sure VCC is within the range pointed out in the device data sheet.

Configurable devices have a fully specified operating range. Reliable operation is only guaranteed within that range.

4. Ensure that VCC is stable during device programming if you are using concurrent configuration strategies.

An adequate power supply is essential for reliable system configuration. Concurrent approaches mean that all devices are erasing and programming simultaneously. Make certain that you system power supply can handle the power needs of the devices during configuration.

5. If you are using long chains of devices with widely distributed TCK and TMS signals, consider building TMS or TCK clock trees as previously discussed.

Attention to design and distribution of these signals is important. Noise induced stray TCK pulses can force a device TAP controller state machine into a different state than all other devices effectively breaking the chain. Therefore, this electrical issue will look like a physical interconnect problem. Debugging and diagnosing these issues can be time consuming.

6. Provide a means to suspend all free running clocks and oscillators during configuration.

System clock signals left running during configuration can be a source of significant electrical noise. You may see coupling of the clock signals to the TAP signal. You may also see a more indirect effect. The clock could be driving some circuitry that drives some state machines that cause signal transitions that couple noise to the TAP signals. This is another time-consuming and tedious problem to track down. It may also be difficult to repair after board fabrication.

7. Make certain to run board tests *before* or *after* device configuration - not *during* configuration.

This circumstance is similar to having free running clocks during configuration. It could be many times worse, though. During test, many signals are activated. The failure noted could be transient or occur at different times depending on the coordination of the configuration and test programs.

8. Provide a means to hold the system in a fixed state during configuration to make it as quiet as possible. This might include providing some system reset signals as well as the previously mentioned free running clock and oscillator controls.

Some devices may respond to initial states produced on programmable device output after configuration completes. For instance, a chip enable signal may be activated. Proper gating of these signals should ensure that no false system operations are started until the system is fully configured or until it is safe to do so.

9. When using IEEE STD 1149.1 or IEEE STD 1532 device chains, group devices with similar logic characteristics together (for example, 3.3Vand 2.5V devices). This reduces the need for special circuitry.

Mixed voltage environments are common. When using devices with mixed IO voltages, you need to do one of the following:

- Provide level shifters between differently powered IOs
- Check that connected devices have IO voltage tolerances suitable for the devices to which they are coupled.
- Connect devices in chains of identical IO voltage levels with level shifter between them.

This last technique is the safest and most reliable.

10. When using IEEE STD 1149.1/IEEE STD 1532 devices, ensure that any compliance enable signals or the TRST TAP signal and easily accessible and controlled by the programming application.

Some devices need special signaling to force them into IEEE STD 1532 (or IEEE STD 1149.1 mode). Access to the pins that enable the TAP is essential to correct device configuration.

The rules related to accessibility of the programming port are relevant only to the development and manufacturing phases. The assumption is that when the final product arrives at the end user site, this access is no longer needed since there will be no further configuration.

There is a separate question about whether this approach is wise. In particular, part of the power of configurable devices is that they allow designers to make changes quickly and potentially late in development or even out in the field. This choice is never available to systems based on ASICs.

If you design a system to be configurable but not *re*configurable then you lose the advantage of applying late breaking fixes at any point in the product's lifetime. You save the cost associated with designing an accessible configuration port but pay the price of increasing the cost for system repair or upgrade.

A block diagram in Figure 11-1 shows a simply configurable system. Note that the configuration port is not accessible after the system is placed in its enclosure. The system can only be practically configured during manufacturing when the board is fully exposed. It may also be the case that there are no posts or connector outlets and that the configuration post is accessible only using a special fixture to contact the port pins.

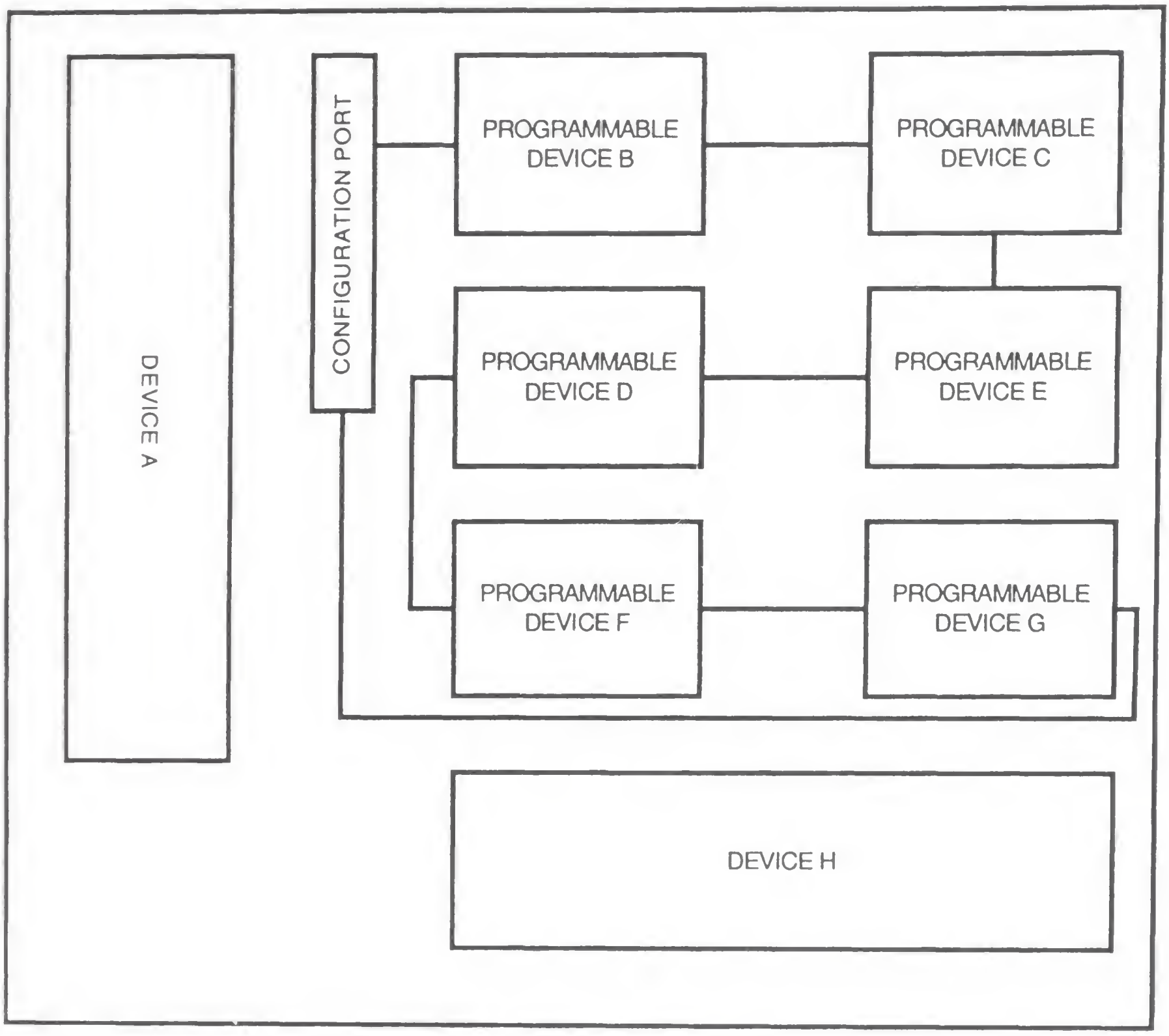

Figure 11-1. Configurable System Block Diagram

3. Designing for Field Reconfigurability

Basic reconfigurability allows for the possibility of reconfiguring a system at any point in its lifetime. A reconfigurable system needs to adhere to all the rules associated with a configurable system. As well, the configuration port must be accessible even after product ship.

This approach is a step up from simple configurability and gives access to the flexibility of programmable logic any time during the system's life cycle. It is worth noting, however, there is a presumption that someone physically present at the target system will perform system updates. There is also the presumption the configuration port is readily accessible by that person. You must knowingly design your systems to allow this access.

Figure 11-2 provides a block diagram of a field reconfigurable system. In this case, the configuration port is placed on the card edge for accessibility after the system is placed in its enclosure.

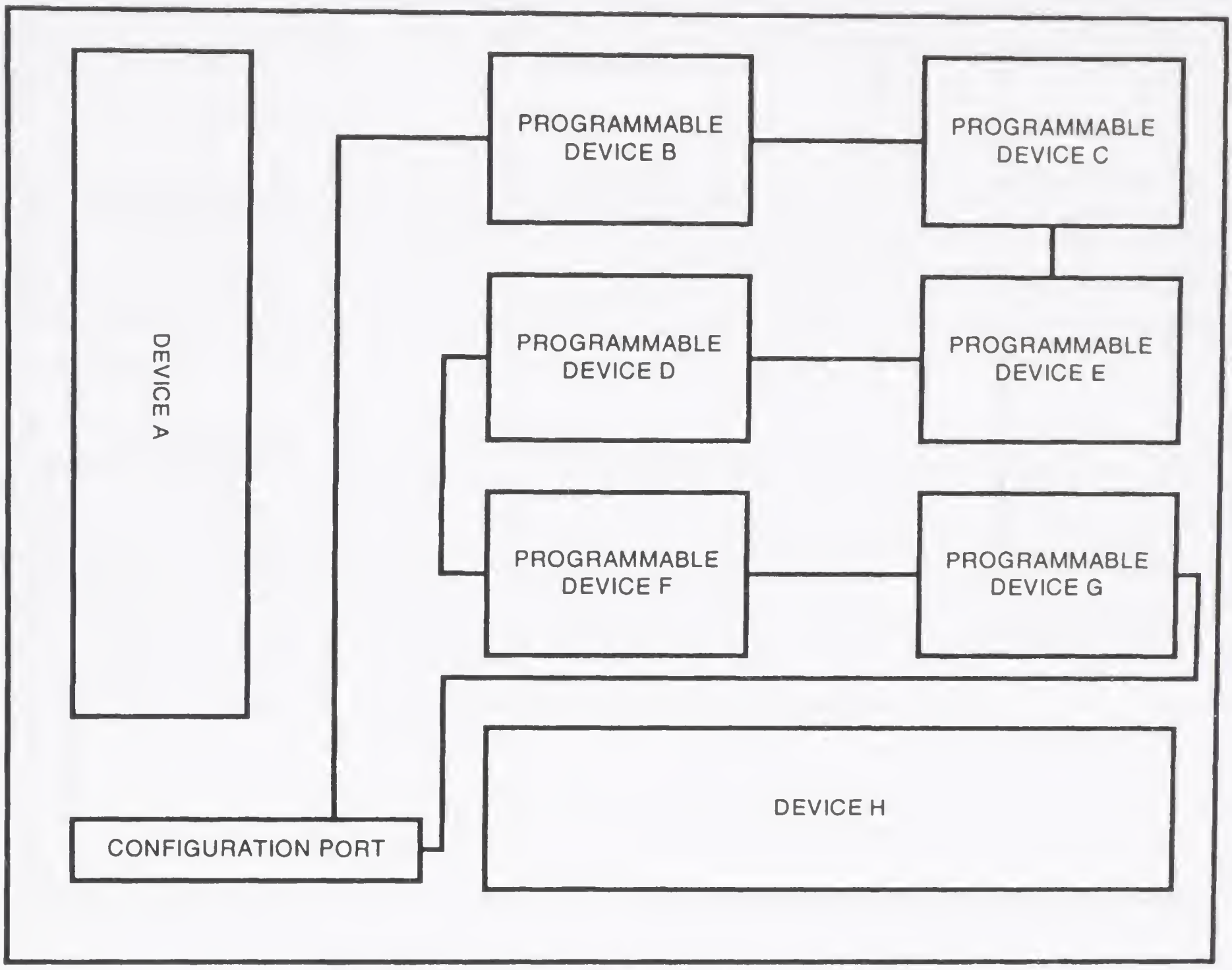

Figure 11-2. Reconfigurable System Block Diagram

3.1 Designing for Network Reconfigurability

The network reconfigurable system builds on the field reconfigurable one by incorporating a method to allow control by a network link. Once a network becomes involved, the overall reliability of the communications method becomes an issue. This means that you must add extra circuitry to ensure that new configuration data received is accurate, complete and usable before allowing the system to use it.

You must always have a default (known working) design available and selectable. There should be a fail-safe method that ensures the system will always come up with a known working function. This need not represent the most up-to-date or most complete functionality. Basic utility is enough to debug, diagnose and resolve any issues. This circuitry should not need operator intervention to function. This is important if the system loses

power, even momentarily, during update. When power is restored and the operator reconnects to the system, the system should always power-up in a usable state with working communications and at least basic system function.

For high reliability operations, there should be a system watchdog function. This is a monitoring application to test system status periodically. If the system is failing, the watchdog should try to alert a repair center sending a message if possible or even lighting a status LED for the service personnel to see. The watchdog may also try to halt the system safely to avoid lasting functional issues and to bring attention to the failure.

In normal execution, operators should be able to set new default designs after a successful system update. This means the system should always have a known-good, fail-safe configuration.

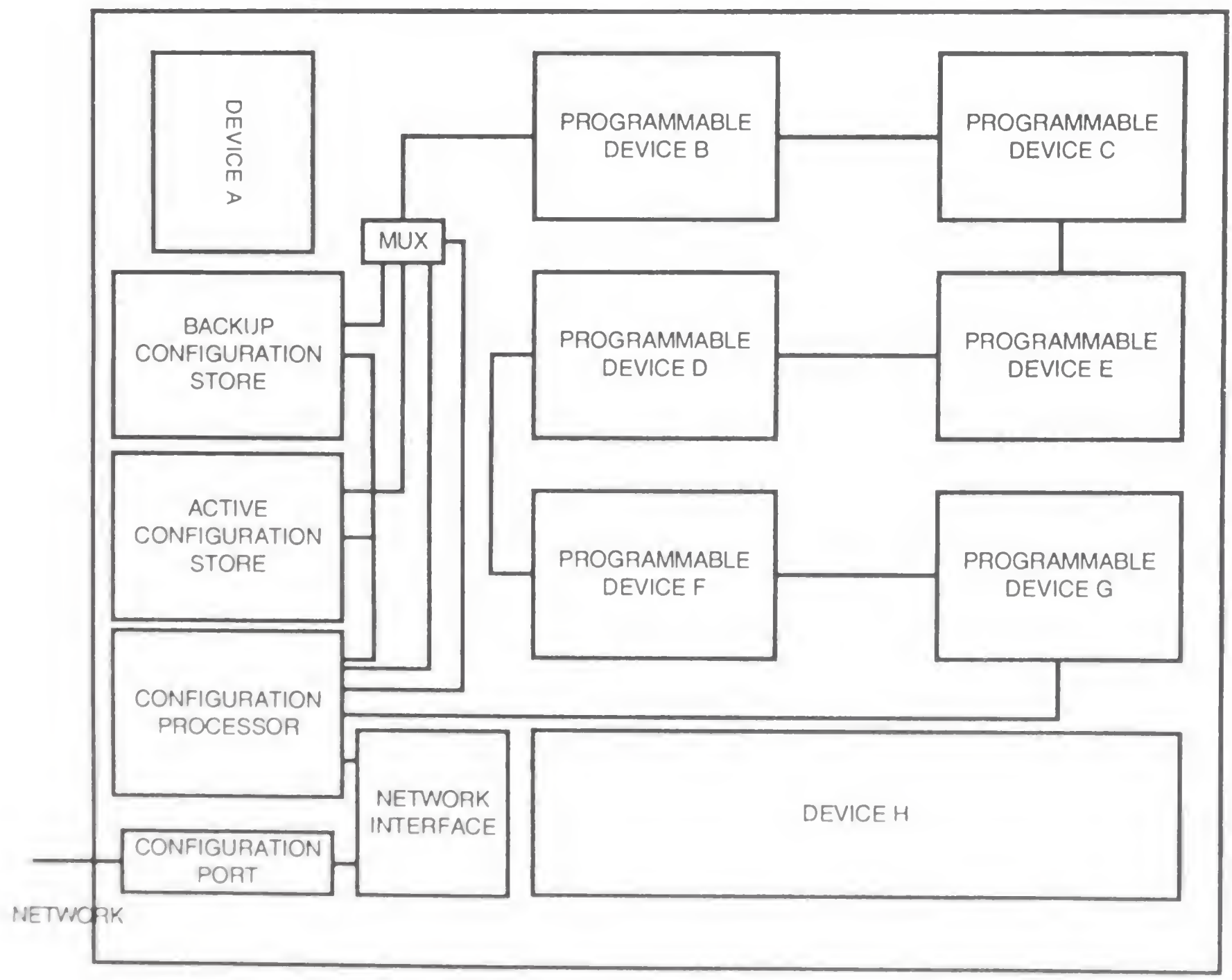

Figure 11-3. Network Reconfigurable System Block Diagram

Figure 11-3 depicts a block diagram of a network reconfigurable system. More functionality is needed of the system that may increase the overall cost

and consume more board space. It is possible to reuse functionality of parts already on the board. For instance, an already included processor and network interface may be used for configuration tasks as well as other mission tasks.

It may be possible to use only one configuration store (rather than the two depicted). An always-correct back-up store remains on board but updates are transferred by the network and not stored locally. This means that when the system reverts to the back-up store (after a power glitch, for instance), a remote operator must intervene to update the system to the latest revision.

The configuration processor controls the flow of configuration data. It can access the programmable devices directly. It also accesses either of the configuration stores and controls which store serves as the configuration data source.

4. Designing For Periodic Reconfigurability

Periodic reconfigurability typically builds on field reconfigurability and allows the system ready access to changing data. The system polls a source location for updates. This source could be removable media, a local disk drive or data stored across a network. Data updates may occur during system operation. Alternatively, before any operation, the system can first poll for updates and then store them locally. Since the configuration changes only now and then, the system will likely use the same functionality for an extended period before changing.

This may be characteristic of systems released to the field for alpha or beta testing and incorporating field update for improvements and fixes.

5. Designing For Frequent Reconfigurability

Often reconfigurable systems change functionality with every invocation. Systems of this sort might change their task not only immediately before beginning execution but also immediately after invocation or completion. They are similar to periodically reconfigurable systems except that these systems may change their functionality from invocation to invocation.

An example of this system may be a communications switch handler that needs to respond to incoming and outgoing traffic with different protocols. Device size limits may require that different protocols be set up as different configuration images. The system might work in the following manner. Initially configured to accept data using protocol A, the device stores the received data in on-chip or off-chip RAM. The data needs to be relayed using protocol B. The device is reconfigured with the image that supports protocol B, reads the data out of the RAM, and sends it out. The device then reconfigures itself to accept protocol A and waits for the next data packet.

This can be abstracted to support any number of different protocols, each one with an associated configuration image.

The configuration images could be stored either, on board in a large memory or even at a remote site and accessed by a network connection. This latter approach is more complicated and may be less reliable since the network connection becomes the weakest link.

6. Designing for Runtime Reconfigurability

A run time reconfigurable system changes its task over the course of carrying out the system function. An example of this is an application that changes during its execution to complete the needed function. These systems must minimize overall system downtime during reconfiguration. This issue then is to design systems that are able to reconfigure quickly. There are many approaches to reach this goal.

6.1 Designing for Rapid Reconfigurability

Rapid reconfigurability focuses on minimizing system down time during reconfiguration. The reconfiguration is a context switch. The system, initially running in one mode or on one task switches to perform a new task. There are several limits to keep in mind. One is the maximum acceptable latency time. That is the maximum time during which the system can be doing nothing while the device is accepting a new configuration. Another is maximum context switching frequency. This signals how often a new configuration is needed. In the worst case, these two values are equal. For example, you must deliver new functionality every 3 minutes and the system can tolerate doing nothing for only 200 msec. For this system to work, the maximum acceptable latency time must be less than the inverse of the maximum context switching frequency. In other words, there must be

enough time to get its configuration data loaded activated before sending the next load of configuration data.

When considering devices for these applications you must understand if they remain active during configuration or not and if so what are they actively doing? A device compliant with IEEE STD 1532 needs to have all programmable pins adhere to the HIGHZ and CLAMP behavior rule of the standard. However, if all the device pins are fixed system pins, they do not follow that rule. That means these pins may be doing something during configuration. It is important for you, the system designer, to understand what they are doing. It had better be something predictable and controllable. When the context switch occurs, you do not want to damage the system. This means the designer must take special care to ensure that context switch is a system-safe operation.

Context switches must therefore occur only when the system is in a known safe state. One method for this is pin state (or pin state sequence) monitoring. This discovers when the system reaches a safe state by following the system pin states. Comparing the pin states (or pin state sequences) against known values (or value sequences) signals a safe state. Then, either the system automatically performs the context switch or it alerts the operator on reaching the safe state. The operator can then perform a context switch under her control.

There could be complications when several devices need to switch as a group. Specific complications could include a requirement of sequencing the context switches of the devices to ensure safety of system operation with each step. Once again, special circuitry can play a role here or the operator can execute instructions to complete the sequence manually.

With these rules in mind, we will consider several approaches to process the configuration data.

The first approach is merely to get the configuration bits into the device as quickly as possible. An example of this is running the IEEE STD 1532 or other serial bit interface at the maximum speed. The drawback of the serial approach is that one bit at a time is loaded. Each clock cycle therefore feeds only one significant bit to the device at a time. In addition, in a serially connected chain of devices the device with the lowest maximum configuration speed controls the system maximum configuration speed. This means that if a single device has a maximum speed of 1 MHz and all other device have a speed of 50MHz the system can configure only at 1 MHz.

These drawbacks can be overcome by putting all like-speeded devices in a single independent chain or even by having each device have its own independent configuration port. While this might decrease the overall system configuration time (and therefore the system down time) it increases the system cost by needing a multiplicity of configuration ports and potentially a more complicated configuration controller to manage system configuration.

Another approach would be to use a byte wide or word wide configuration port (if available on the device). These ports typically deliver 8 (or sometimes 16 or 32) bits of configuration data at a clock rate similar to that of the serial mode, increasing throughput by 8 (or 16 or 32). These interfaces are typically point-to-point. An example of this methodology is included as Figure 11-4. This simplified block diagram includes a network interface to receive the configuration data from a remote location. This is optional functionality. The heart of the system is the configuration processor that selects the target programmable device. Typically a single device can be configured at a time. However, if the configuration data is identical for all devices then all may be configured simultaneously.

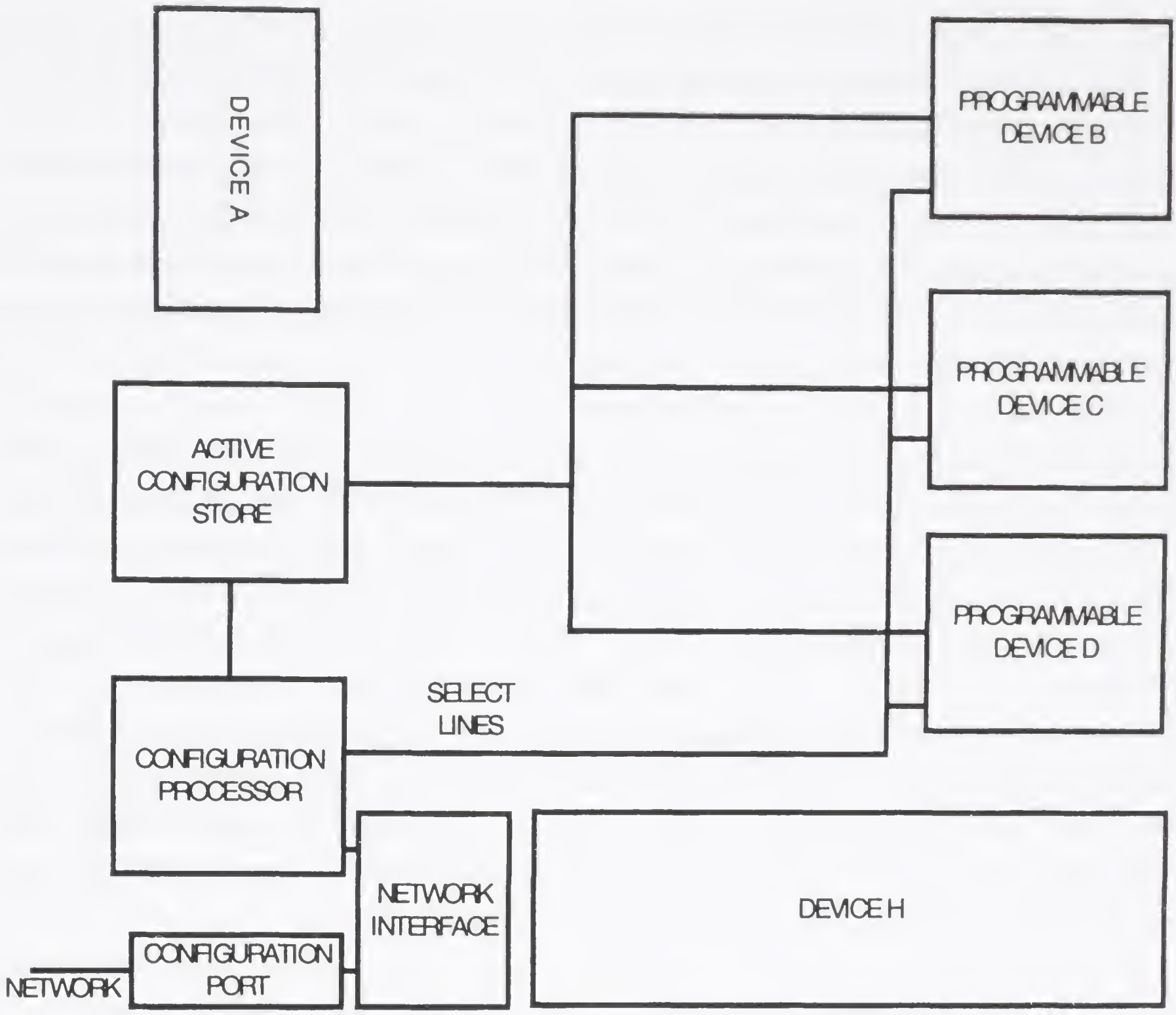

Figure 11-4. Rapid Reconfigurable System Block Diagram -Parallel

Yet another approach would be to apply a partial reconfiguration method to reconfigure the system incrementally canceling system downtime in localized areas of functionality. This would involve a phased and controlled shutdown and startup of the system. Of course, not all devices support partial configuration so this technique would be device specific.

A final consideration would be the use of on-the-fly techniques. That reduces the system down time to the device activation time. It may in fact be the case the configuration time doesn't matter when using on-the-fly reconfiguration since the down time is so small, you can transfer configuration data while the system is running. You should be certain the context switching time does not exceed the time it takes to configure the device in the background before the switch.

There is a variation of on-the-fly reconfigurability for volatile devices. We will use Xilinx's Virtex2PRO family of FPGAs to provide an example. The Virtex2PRO devices have an integrated PowerPC (PPC) processor and a

special port internal to the device called the Internal Configuration Access Port (ICAP). The ICAP allows access to the device's configuration memory. You can use the PPC to read and write the configuration memory and effect small changes to the device function, quickly.

For instance, if you need to change the device IO characteristics from LVDS to LVPECL, the configuration patterns to perform this are small and easily stored in on or off-chip memory. The PPC can read and apply this configuration information and rapidly change the IO characteristics. This does not happen instantaneously like on-the-fly mode in nonvolatile devices but it does happen quickly.

7. Summary

Reconfigurable systems occupy a wide range. Designers need to consider reconfiguration early in the design process to ensure efficient and manageable implementation. Before choosing a reconfiguration strategy, designers need to be aware of the reconfiguration needs of their system.

Chapter 12

Conclusion

Programmable logic enabled developing a new class of systems that in themselves are programmable. With the approval and acceptance of IEEE STD 1532, device vendors essentially agreed the configuration algorithm itself is not protected intellectual property. The value of the device was its logic functionality and its ability to be used easily within a system.

IEEE STD 1532 strengthens this latter feature. IEEE STD 1532 relegates developing customized configuration applications to the dustbin and allows configuration to be more easily included as general-purpose system functionality. However, it does not free the designer from designing the configuration infrastructure.

The rules are simple. First, decide on the class of configurable system you are designing. Where is your application on the spectrum of configurability? Then, when you have resolved that issue, ensure that you design in the programmable and support components needed to carry out the functionality chosen. This includes ensuring that and additional components to control configuration are identified and incorporated, as well as designing or assembling the software required. Finally, you must create the configuration port network. It is essential that it, too, be designed - and not simply allowed to happen.

This latter point cannot be over-emphasized. Too many systems, even simple ones, either consistently or sporadically fail configuration because of bad configuration network design. Therefore, Chapter 11 is the most important chapter in the book.

This book was designed to be both useful and practical in nature and serve as a reference for developing in-system configurable systems of the present and the future. I hope it has achieved these goals.

References

"A Quick JTAG ISP Checklist." Xilinx, Inc. June 7, 2002.

"Accelerator Series FPGAs - ACT 3 Family." Actel, Inc. September, 1997.

"ACT 1 Series FPGAs." Actel, Inc. April, 1996

"ACT 2 Family FPGAs." Actel, Inc. December, 2000.

"APEX II Programmable Logic Device Family Data Sheet", Altera, Inc. August, 2002.

"CoolRunner II CPLD Family." Xilinx, Inc. March 12, 2003.

"CoolRunner XPLA3 CPLD Family." Xilinx, Inc. June 23, 2003.

"FLEX 8000 Programmable Logic Device Family Data Sheet.", Altera, Inc. June, 2003.

"Introduction to JTAG and Five-Volt Programming with In-circuit Programmable MACH Devices." Advanced Micro Devices, February, 1996.

"ispMACH 4A Family Architectural Description." Lattice Semiconductor Corporation. June, 2000.

"ispMACH 5000B Family Data Sheet." Lattice Semiconductor Corporation. September, 2002.

"ispVM EMBEDDED Software." Lattice Semiconductor Corporation, December, 2001.

"ispVM System Software." Lattice Semiconductor Corporation, December, 2001.

"Frequently Asked Questions (FAQ) About Programmable Logic." The Programmable Logic Jump Station. OptiMagic, Inc. Retrieved June 20, 2003 <www.optimagic.com/faq.html>

"MAX 7000 Programmable Logic Device Family Data Sheet.", Altera, Inc. June, 2003.

"MAX 9000 Programmable Logic Device Family Data Sheet.", Altera, Inc. June, 2003.

"Quicklogic ESP and FPGA Data Book." Quicklogic, Inc. December, 2000.

"Stratix Device Handbook." Volumes 1-3, Altera, Inc. July, 2003.

"Virtex 2.5V FPGA Complete Data Sheet." Xilinx, Inc. December 9, 2002.

"VirtexE 1.8V FPGA Complete Data Sheet." Xilinx, Inc. March 14, 2003.

"XC4000XLA FPGAs Description." Xilinx, Inc. October 18, 1999.

"XC5200 FPGAs." Xilinx, Inc. November 5, 1998.

"XC9500 5V In-System Programmable CPLD Family." Xilinx, Inc. September 15, 1999.

"XC9500XL 3.3V High Performance CPLD Family." Xilinx, Inc. January 24, 2002.

"Xilinx In-System Programming Using an Embedded Microcontroller." Xilinx, Inc. January 15, 2001.

"IEEE STD 1149.1-2001: IEEE Standard Test Access Port and Boundary-Scan Architecture". New York, NY: The Institute of Electrical and Electronics Engineers, 2001.

"IEEE STD 1532-2002: IEEE Standard for In-System Configuration of Programmable Devices". New York, NY: The Institute of Electrical and Electronics Engineers, 2003.

"Virtex II Complete Data Sheet." Xilinx, Inc. June 19, 2003.

"Virtex II Pro Complete Data Sheet." Xilinx, Inc. June 19, 2003.

"ispLSI 3256 Data Sheet." Lattice Semiconductor Corporation. June, 1999.

Bleeker, Harry, Peter van den Eijnden, Frans de Jong. "Boundary-Scan Test". Dordrecht, The Netherlands: Kluwer Academic Publishers, 1993.

Brown, Stephen and Jonathan Rose. "Architecture of FPGAs and CPLDs: A Tutorial". *IEEE Design and Test of Computers*, Vol. 13, No. 2, pp. 42-57, 1996.

Campione, Mary, Kathy Walrath, Alison Huml. "The Java Tutorial: A Short Course on the Basics". 3rd Edition, Addison-Wesley Publishers, 2000.

Cappaletti, Paulo, Carla Golla, Piero Olivo, Enrico Zanoni. "Flash Memories". Kluwer Academic Publishers, 1999.

Chen, Zhiqun. "Java Card Technology for Smart Cards: Architecture and Programmer's Guide". Addison-Wesley Publishers, 2000.

Coombs, Clyde F. "Coombs' Printed Circuit Handbook". 5th Edition, McGraw Hill Professional, 2001.

Dorf, Richard C., Editor. "The Electrical Engineering Handbook". 2nd Edition, CRC Press, 1997.

Fink, Donald G., Wayne Beaty, Editors. "Standard Handbook for Electronic Engineers". 14th Edition, McGraw Hill Professional, 1999.

Ganssie, Jack. "The Art of Designing Embedded Systems". Woburn, MA: Newnes Press, 2000.

Jenkins, Jesse H. "Designing with FPGAs and CPLDs". Prentice Hall, 1994.

Lipman, Jim. "Growing Your Own Clock Tree". *EDN*, March 14, 1997.

Parker, Kenneth P. "The Boundary-Scan Handbook". Second Edition. Norwell, MA: Kluwer Academic Publishers, 2000.

Sharma, Ashok. "Advanced Semiconductor Memories: Architectures, Designs and Applications". Wiley IEEE Press, 2002.

Sharma, Ashok. "Programmable Logic Handbook: PLDs, CPLDs and FPGAs". McGraw Hill Professional, 1998.

Trimberger, Steve, Editor., "Field Programmable Gate Array Technology". Norwell, MA: Kluwer Academic Publishers, 1994.

Index

The In-System Configuration Handbook: *A Designer's Guide to ISC*

Programmable logic radically changed the electronic system design landscape. It reduced board space needed for random logic, state machines and system interfaces. It allowed faster design cycles, made easy late term bug fixes and gave designers greater freedom to experiment and prototype.

In-system programming of these devices has had a similar revolutionary effect. The ability to change the programmed content of programmable logic while it is on the board is equivalent to being able to redesign all the hardware—without changing a single component. This allows the possibility of providing field upgrades of your product to fix problems or to introduce new functionality. It allows designing in reconfiguration as an essential function of your system with different capabilities swapped in as needed during run-time. Further it allows storage of different product profiles for retrieval as necessary to allow just-in-time configuration of systems to meet market needs.

Recent developments in programmable logic have helped to make reconfigurable systems more streamlined. The most significant development, however, was the introduction, approval and popularization of IEEE STD 1532, the IEEE Standard for In-System Configuration of Programmable Devices. While focusing on IEEE STD 1532, this book surveys all of the available techniques and products that ease the development of in-system configurable applications. In addition, The **In-System Configuration Handbook: *A Designer's Guide to ISC*** provides design considerations and rules-of-thumb to ensure that the functionality you want will work.

The purpose of this text is to bring together, in a single volume, the information needed by systems designers to develop applications that include configurability. This covers the entire range of systems from the simplest implementations that merely include configurable logic to realize system functions to the most complicated that include reconfigurability as part of the application itself.

This book is written for IC Designers, System Designers and Test Engineers.

ISBN 1-4020-7655-X

9 781402 076558

Kluwer Academic Publishers
1-4020-7655-X